AN FPGA IMPLEMENTATION OF ATR USING EMBEDDED RAM FOR CONTROL

Richard D. Ross

Department of Electrical and Computer Engineering

M. S. Degree, June 30, 1997

ABSTRACT

Automatic Target Recognition (ATR) is a computationally intensive problem with potential for good performance when mapped to Field Programmable Gate Arrays (FPGAs). This thesis presents work that was done to implement the Sandia National Laboratory Chunky SLD stage of ATR on an Altera FLEX 10K50. The FLEX 10K series has large (256×8), dedicated, embedded memories that present an opportunity for unique and innovative implementations of computing algorithms. These memories were used for several purposes; the most interesting use was to store microinstructions that direct the operation of the ATR processor. With this method of implementing Chunky SLD, good performance was achieved relative to other FPGA and microprocessor implementations.

COMMITTEE APPROVAL:

Brad L. Hutchings, Committee Chairman

James K. Archibald, Committee Member

Brent E. Nelson, Committee Member

Brent E. Nelson, Department Chairman

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
30 JUN 1997		**00-00-1997 to 00-00-1997**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
An FPGA Implementation of ATR Using Embedded Ram for Control	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Brigham Young University,Department of Electrical and Computer Engineering,Provo,UT,84602**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**129**	

AN FPGA IMPLEMENTATION OF ATR USING EMBEDDED RAM FOR CONTROL

A Thesis
Presented to the
Department of Electrical and Computer Engineering
Brigham Young University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Richard D. Ross
June 30, 1997

This thesis by Richard D. Ross is accepted in its present form by the Department of Electrical and Computer Engineering of Brigham Young University as satisfying the thesis requirement for the degree of Master of Science.

Brad L. Hutchings
Committee Chairman

James K. Archibald
Committee Member

Brent E. Nelson
Committee Member

Date

Brent E. Nelson
Department Chairman

ACKNOWLEDGMENTS

There are many people I would like to acknowledge and thank. First, this work was funded by DARPA/CSTO. Without funding, none of this work would have been done. Next, I want to thank Dr. Brad Hutchings for his help both with the project itself and with the reviewing of the thesis. Many of the ideas for the design are his. Thanks also go to Dr. Brent Nelson and Dr. Jim Archibald for their time and effort in reviewing my writing. Thesis reading can be pretty dry sometimes. The guys in the configurable computing lab were also a help in many ways. In particular, I want to thank Mike Wirthlin, Paul Graham, Mike Rencher, and Justin Diether.

Last, but of course not least, I express my appreciation to my wife Angie. She has been very patient, supportive, and sacrificing through the whole process. Hopefully, when this is over, she'll have a husband again.

Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

With the continual advances in VLSI technology, applications that were once too difficult for computing hardware to handle are becoming more and more feasible. One such application is Automatic Target Recognition (ATR). ATR is a data intensive algorithm that is of great interest to the military. Although ATR can be implemented on general purpose microprocessors, the performance of these processors is not good enough to make their use feasible in a real system. This has lead researchers to investigate the use of reconfigurable logic for use in this area. Field Programmable Gate Arrays (FPGAs), which are the principle reconfigurable logic devices, show great promise for being able to handle the high data flow and computation necessary to implement ATR in a real, usable system.

1.1 Automatic Target Recognition

Automatic target recognition (ATR) is an application from the field of pattern recognition. Very simply, ATR involves searching through images with a computer in search of an object, or target, of interest. Pattern recognition is essentially the same thing with the difference being that ATR usually refers to a military application of pattern recognition. The images may come from RADAR or satellite photographs or elsewhere. The objects are things the military is interested in identifying.

The ‘‘Automatic’’ in ATR refers to the fact that a computer is doing the target location instead of a person. The reason a computer is used is not to increase accuracy. Humans are actually very good at pattern recognition, that is, we are relatively accurate in locating objects in images. A computer is used to speed up the process. In many situations where target location is required, there are too many images and too many objects of interest for a human to handle. For example, the ATR algorithm presented shortly requires that an image be searched for a minimum of 100 targets per second. This high data rate is required because of the large number of targets of interest and the large number of images that have to be searched.

ATR does not replace a human operator completely. It just attempts to find the most likely matches in images and point them out to the operator so he can make a final decision. In this way it acts as a ‘‘weeding out’’ process or data filter.

There are various ways to perform automatic target recognition. The algorithm used here is one that was developed at Sandia National Laboratory. It is discussed in detail in Chapter 2. Throughout this report, the distinction between the‘‘algorithm’’ and the ‘‘implementation’’ should be kept in mind. The algorithm was developed at Sandia National Laboratory. It is the mathematical description of the operations performed. The implementation that will be described was developed at BYU as part of this project. It is the method used to realize the algorithm in hardware.

1.2 Field Programmable Gate Arrays

As the name implies, an FPGA is a type of programmable logic device. Another term to describe FPGAs is reconfigurable hardware. In essence, an FPGA is a piece of hardware whose internal functional structure can be modified by its user. Contrast this with a microprocessor whose function is fixed. Although a microprocessor can be made to perform different tasks through software programming, its internal structure never changes. It always has the the same functional blocks and the same wiring between those blocks.

An FPGA does not generally change in its physical structure, but its functionality can be modified in such a way that it appears to the user as if the physical structure has changed. This is accomplished in various ways. The SRAM FPGA incorporates one of the most popular methods. An SRAM FPGA is made up of thousands of static memory cells that control the functionality of the device. The FPGA can be reconfigured to perform different tasks by changing the contents of the memory cells. Because reconfiguration only involves loading new data into memory cells, it can be done quickly and an unlimited number of times.

By far the largest use of FPGAs today is to implement various logic functions that tie together other components in a system. This is called gate replacement, ‘‘jelly bean logic’’ or ‘‘glue logic.’’ FPGAs are well suited to this and are used widely in industry for this purpose. In recent years, however, researchers have been studying the feasibility of more ambitious uses for FPGAs. In particular, they have been looking at

ways to use an FPGA in a system either in place of or in addition to a microprocessor. The hope is that the reconfigurable logic will increase the performance of the system. This area of research is called configurable computing. ATR is an example of an application that fits into this category. It is more than just gate replacement, it is actual computation. It is something that a microprocessor might be used for, but current microprocessors do not perform well enough to use in a real system. Because of the characteristics of the algorithm, however, it is well suited to configurable hardware.

1.3 Project Objectives

There are many possible ways to implement ATR on FPGAs. Several implementations have been and are being studied at Brigham Young University [1, 2]. What makes the implementation presented here unique is shown in the objectives of the project, which were to

- exploit the sparseness of the bright templates,
- utilize the on-chip RAM of the Altera FLEX 10K FPGA, and
- demonstrate the use of embedded RAM for control.

The terms in used in these objectives will be explained more in later chapters.

1.4 Overview

The presentation is as follows. Chapters 2 and 3 give the necessary background on ATR and the Altera FPGA. Chapter 4 explains this project's implementation for a single ATR processor and Chapter 5 shows how the implementation is extended to multiple processors running in parallel. The performance of the implementation presented here depends on the characteristics of the templates (templates are explained in Chapter 2) so Chapter 6 explains how template data was synthesized to predict performance. Chapter 7 gives the results of the hardware synthesis, Chapter 8 discusses the issues involved in scaling the design to larger devices, and Chapter 9 summarizes and draws conclusions.

Chapter 2

AUTOMATIC TARGET RECOGNITION

The ATR algorithm used in this project was developed at Sandia National Laboratory. It is described in this chapter. The next chapter discusses the implementation that was developed at BYU. The algorithm is made up of the three major stages shown in Figure 2.1. These stages are Focus of Attention (FOA), Second-Level Detection

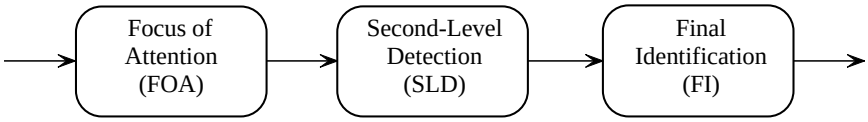


Figure 2.1: Stages of ATR

(SLD), and Final Identification (FI). Each of the stages narrows the search down to a smaller region so that succeeding stages which perform more computations process less data. In this way the stage requiring the most computation, final identification, has to process only a fraction of the original data.

This project deals only with the SLD stage of the algorithm. Work on other stages is currently being done at the Configurable Computing Laboratory at Brigham Young University. Work on SLD has been done elsewhere with good success (see [3]). A variation on SLD, called *chunky SLD*, is actually what was implemented for this project. It is similar in some ways to normal template matching by cross correlation [4]. This chapter discusses the data for chunky SLD and the calculations that are performed on that data.

2.1 Chunky SLD Templates and Images

As with normal template matching, images are compared to templates to find possible matches. The templates are themselves small images that represent the objects or patterns that are being searched for in an image. The template is placed at different

positions in the image, and at each of these positions, the region of the image over which the template lies is checked for a match.

2.1.1 Image and Template Size

For this version of ATR, the images come from synthetic aperture radar, or SAR. Their size changes from stage to stage. In the FOA stage, they are large (1024×896), but for the chunky SLD stage they are 128×128 pixels with each pixel having eight bits of depth. For chunky SLD the image is sometimes called an *image chip* or just a *chip.*

The templates represent different parts or ‘‘chunks’’ of the targets. This is where the term chunky comes from in chunky SLD. The algorithm is intended to find partially obscured objects by searching for parts of the object in the image chip. The templates are themselves binary images, meaning each ‘‘pixel’’ is really just one bit that is either on or off. The size of these templates is 16×16. Figure 2.2 shows the relative size of the image chips and templates for chunky SLD.

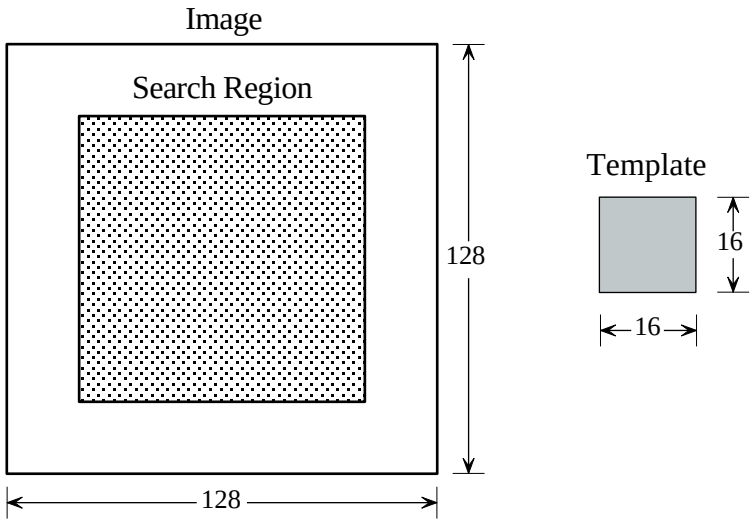


Figure 2.2: ATR Data

There is a pre-defined region within the image chip over which the template matching is done. This region is an area 65×65 in the image. It is referred to as the

search region.

2.1.2 Template Hierarchy

Many templates are required to represent one target in chunky SLD. There are three major reasons for this. First, recall that chunky SLD attempts to identify partially obscured objects. To do this, the target is divided into pieces, or chunks, and each chunk is represented in a template. That way, if part of the target is behind cover, some subset of the templates may still match the exposed part of the target.

The second reason that many templates are required is that the target may be facing any direction. A target facing east requires completely different templates to identify it than a target facing north or anywhere in between.

The third complication is that each chunk discussed above actually requires two templates. One template, the *bright* template, contains pixels where strong RADAR return is expected. The other, the *surround* template, contains pixels where strong RADAR absorption is expected. The templates are grouped together in large hierarchies where each hierarchy contains all the templates necessary to completely detect one target. Each hierarchy is called a ‘‘Q’’. Each Q has the templates for all the rotations of a target, all the chunks at each rotation, and both the surround and the bright templates for each chunk.

Figure 2.3 shows this hierarchy, along with the number of templates for each level. At the lowest level of the Q are the bright and surround templates. One pair of a bright and a surround template is called a chunk. Forty chunks together form a *class.* A class represents a target at one orientation. Each object is represented with 72 orientations so 72 orientations of a class make up a Q. This gives $72 \times 40 \times 2$ or 5760 templates in a Q, or 2880 chunks.

2.1.3 Data Characteristics

In bright templates typically only a small percentage of the pixels are on. This important feature is exploited in this implementation of chunky SLD. In contrast, in surround templates a large majority of the bits are on. Figure 2.4 shows an example of a typical bright and surround template pair.

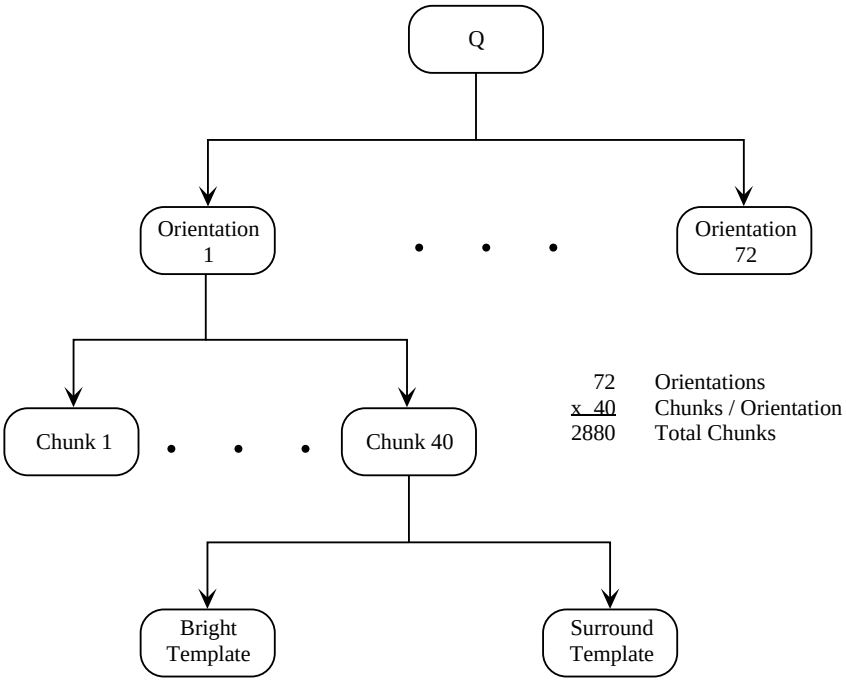
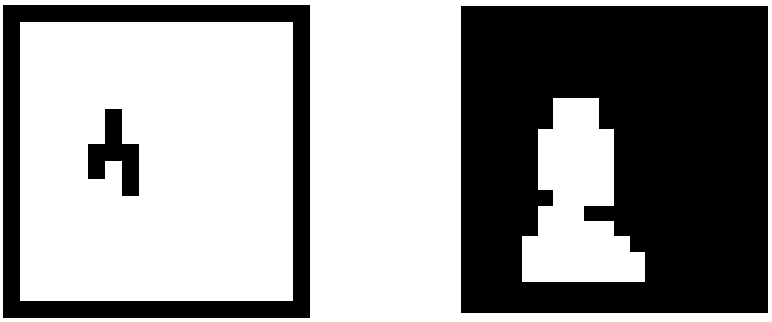


Figure 2.3: Template Hierarchy



Bright Template Surround Template

Figure 2.4: Example of Typical Templates

The template data comes from a database which is static, and thus is known at compilation time. This gives some flexibility as to how the template data is stored and accessed. The images, however, are available only at run time. Nothing about them except for their size is known before-hand.

2.2 ATR Algorithm

There are several operations performed in chunky SLD. They consist of the

- Computation of the shapesum,
- Thresholding of the image,
- Computation of the brightsum and surroundsum, and
- Computation of the final hitcount.

Figure 2.5 shows how these operations are linked together. These operations are

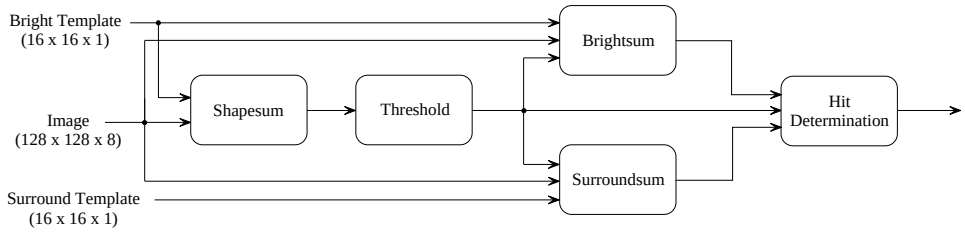


Figure 2.5: Chunky SLD Stage

performed with each chunk at all pixel positions in the image that are within the *search region* (Figure 2.2). That is, the bright and surround templates under consideration are placed at a position within the search region of the image, the operations are performed, then the templates are moved to the next position in the search region. The following sections explain the details of the operations for a single offset in the image.

2.2.1 Shapesum Calculation

The shapesum operation is a correlation between the bright template and the image. Each bit in the bright template is multiplied by the image pixel that lies under it,

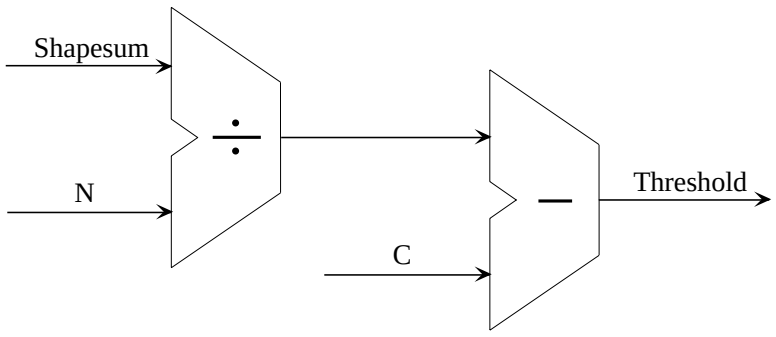
and the results are summed. That is,

$$ssum(x, y) = \sum_{a,b=0}^{15} btemp(a, b) \times im(x + a, y + b), \tag{2.1}$$

where $ssum(x, y)$ is the shapesum for a particular offset (x, y) in the image, $btemp(a, b)$ refers to the bit in the template at the offset (a, b), and $im(x + a, y + b)$ is the pixel in the image chip at location $(x + a, y + b)$.

2.2.2 Threshold Calculation

The threshold is calculated from the shapesum. It is used in the brightsum and surroundsum calculations. To calculate the threshold, the shapesum from the previous step is divided by the number of on-bits in the template, and then a constant is subtracted from the result of the division. Figure 2.6 shows a block diagram of this operation. The



N = number of on-bits in template
C = template constant

Figure 2.6: Threshold Calculation

constant in this calculation is actually part of the template data and associated with each bright-surround template pair.

2.2.3 Brightsum Calculation

The brightsum operation uses the bright template data, the image data, and the threshold that was calculated previously. The brightsum operation consists of the following steps:

1. Each bright template bit is multiplied by the image pixel under it (since the template is binary, the multiplication reduces to an ‘‘AND’’ operation).

2. The result is compared to the threshold that was calculated for this image offset.

3. A counter is incremented if the multiplication result is greater than the threshold.

In equation form:

$$bsum(x,y) = \sum_{a,b=0}^{15} btemp(a,b) \times [im(x+a, y+b) \geq thr(x,y)]. \tag{2.2}$$

The variables have the following meanings:

- $bsum(x,y)$ is the brightsum value at offset (x,y) in the image.
- $btemp(a,b)$ is the bit at offset (a,b) in the bright template.
- $im(x+a, y+b)$ is the image pixel at offset $(x+a, y+b)$.
- $thr(x,y)$ is the threshold value for the image offset (x,y)
- $[im(x+a, y+b) \geq thr(x,y)]$ returns a 1 if true and a 0 if false.

The result of this operation is a count of the number of pixels that are under on-bits in the template and that are greater than the calculated threshold.

2.2.4 Surroundsum Calculation

The surroundsum operation is very similar to the brightsum operation. Each bit in the surround template is multiplied by the image pixel under it. The result is compared to the threshold (the same threshold as for the brightsum), and a counter is incremented if the multiplication result is less than the threshold. The differences are that the surround template is used instead of the bright template, and the counter is incremented when the computed value is less than the threshold. In equation form, the surroundsum is given by

$$surrsum(x,y) = \sum_{a,b=0}^{15} surrtemp(a,b) \times [im(x+a, y+b) \leq thr(x,y)]. \tag{2.3}$$

2.2.5 Hit Calculation

A hit is obtained if the threshold, brightsum, and surroundsum meet certain criteria. The criteria are that $(thresh_{min} \leq thresh \leq thresh_{max})$, $(brightsum \geq brightsum_{min})$, and $(surroundsum \geq surroundsum_{min})$ where $thresh_{min}$, $thresh_{max}$, $bright_{min}$, and $surround_{min}$ are constants that are part of the template data. The hits from all the templates in a class are counted to calculate a hitcount which is used in the next state of ATR.

Chapter 3

ALTERA 10K FPGA

The Configurable Computing Lab at BYU is and has been investigating ATR implemented on FPGAs from several different manufacturers. The device used for this particular implementation is the Altera FLEX 10K. This FPGA comes in several sizes with varying resources.

3.1 Embedded RAM

The most interesting feature of the Altera 10K family is its large (2 Kbit) embedded RAMs called EABs (Embedded Array Blocks). These EABs are dedicated RAM resources on the device. They are not logic cells configured as RAM. Other FPGA makers, such as Xilinx, allow configuration of FPGA logic cells to act as embedded RAM. The Xilinx devices do not have resources dedicated to RAM only. The advantage of the dedicated RAM on the Altera parts is that they are fairly dense and have a high storage capacity. The disadvantage is that if they are not needed as RAM, they cannot be used as normal logic cells. As was mentioned in the introductory chapter, one of the goals of this project was to exploit these embedded RAMs in some way to achieve high performance.

3.1.1 Uses for Embedded RAM

Embedded RAM in FPGAs has many uses. This certainly is not the first project to benefit from such RAM. Embedded RAM has been used to implement

- multipliers,
- complex logic,
- transcendental functions,
- data buffers, and

- state machine decoding [5].

Generally, EAB usage falls into three categories:

- look-up tables,
- buffers, or
- control.

When used as a look-up table (LUT), embedded memory can replace large sections of combinational logic in a design. For example, a 256×1 memory can perform any logic function of eight inputs. The eight inputs become the address lines to the memory, and the memory output is the function value. Replacing combinational logic in this way can both reduce the area of a circuit and increase its speed.

When used as a buffer or other similar small storage device, embedded memory can replace slower, off-chip RAM. This again may speed up the circuit, and it may also reduce the I/O pins that the circuit requires. Since I/O pins are a valuable resource in FPGA designs, this is a real gain.

When used for control, it is typically used to implement state decoding or something similar, which can also be classified under the LUT use. One of the unique aspects this project is that it uses the embedded RAM for control in more creative ways.

3.2 10K Embedded RAM

Each EAB in the Altera 10K has a capacity of 2048 bits. The EAB is flexible in its data and address widths and can be configured in the following sizes: 256×8, 512×4, 1024×2, and 2048×1. Multiple EABs can also be chained together to create larger memories.

The EABs are flexible in other ways also. They can be configured as fully synchronous, fully asynchronous, or somewhere in between. They can be configured as RAM or ROM. In either case a file may be created that contains the initial data that is to be stored in the EAB. This initialization file is required for the ROM configuration. One of the limitations of the EABs is that they cannot be read and written in the same cycle. It is often desirable to read a value from a buffer and replace it with a new value, and this requires two cycles on the FLEX 10K.

3.3 10K Logic and Routing Resources

The logic resources on the 10K are fairly basic. The logic cell is called a logic element or LE. Each LE has a four-input LUT, for combinational logic, and a flip-flop. The flip-flop has asynchronous set and reset, and synchronous enable. The routing consists mainly of long wires that cross the die. This interconnect is called FastTrack. There are also fast carry chains and cascade logic chains between logic elements. The I/O pins are bidirectional with registers and tri-state capabilities.

3.4 10K Capacity

The FLEX 10K comes in several sizes. Table 3.1 lists the resources available for the various available parts. The information in the table is taken from the Altera data book [5].

Table 3.1: Altera FLEX 10K Resources

Resource	10K10	10K20	10K30	10K40	10K50	10K70	10K100
Logic Elements	576	1152	1728	2304	2880	3744	4992
RAM Bits	6144	12,288	12,288	16384	20,480	18,432	24,576
EABs	3	6	6	8	10	9	12
Registers	720	1344	1968	2576	3184	4096	5392
User I/O	150	198	248	278	310	358	406

Chapter 4

SINGLE CHUNK PROCESSOR

This chapter and the next present the implementation of the chunky SLD algorithm. The implementation was developed at BYU for this project. This chapter describes the single chunk processor while chapter 5 extends the implementation to multiple chunk processors. The method discussed here will be referred to as the ‘‘microcode’’ method.

4.1 Shapesum Implementation

As Chapter 2 explained, the shapesum is a cross-correlation between the bright template and the image. The formula for cross-correlation is straight-forward, and it might seem natural to make the implementation closely resemble the formula. This would mean multiplying each pixel in the image by the corresponding bit in the template and summing the results.

While this would certainly accomplish the task, it would not be the most efficient method given the particular conditions in chunky SLD. The characteristics of the templates make several optimizations possible. Specifically, because the templates are binary and sparsely populated (few on-bits), the amount of unnecessary computation can be greatly reduced. This is discussed in the following sections.

4.1.1 Optimizing the Multiplication

Using the fact that the templates are binary images, the multiplication can be performed with a simple AND operation. The template bit is ANDed with the incoming image pixel and the result is accumulated. Another way to do this is to use the bit from the template as a control bit that determines whether the accumulator adds the incoming pixel to the accumulated sum.

4.1.2 Optimizing for Template Sparseness

This simplifies the multiplication, but it doesn't take advantage of the sparseness of the bright templates. Since there are relatively few on-bits in the bright templates, the accumulator is idle a high percentage of the time while the off-bits are being processed. This wastes the FPGA resources dedicated to the accumulator.

To keep the accumulator busy, a pixel needs to be accumulated every cycle. This means that only pixels in the image that lie under on-bits in the template should come into the FPGA. This requires two things: 1) the image must be available in an off-chip RAM so that any pixel can be accessed, and 2) an address must be generated every clock cycle to a ''valid pixel'', that is, a pixel that lies under an on bit in the template.

The first requirement is not unreasonable; random access of the image pixels is possible if the image is stored in RAM. The second requirement means that the bright template must be stored in some manner so that the address generation is possible. This is where the Altera EAB comes in. It is used to store the template data in a format that allows direct access to the on-bits.

If the shapesum operation were performed as a straight-forward cross correlation, the template would be viewed as a small image and stored in a 2-D array, or some similar representation. Finding the on-bits in an array, however, would require a full search of all the bits in that array. Instead of storing the bits of the template in this way, the template structure is stored as a series of offsets to the on-bits of the template. A location in the template is arbitrarily chosen as the reference location. Each on-bit has an x and a y offset from the reference location. The x offset and the y offset for each on-bit are concatenated and stored in one location in the EAB. This is illustrated in Figure 4.1 with the upper left-hand corner as the reference location. The EAB data must be wide enough to store the offsets. Since the templates are 16×16, the x and y offsets each require four bits. The size of a full offset then is $4 + 4 = 8$ bits.

These offsets are then used to calculate addresses to pixels that are required for the shapesum calculation. This is done by adding each offset, one at a time, to a base offset in the image. This base offset is the location in the image for which the shapesum operation is being performed. The base offset plus the template offset gives the address

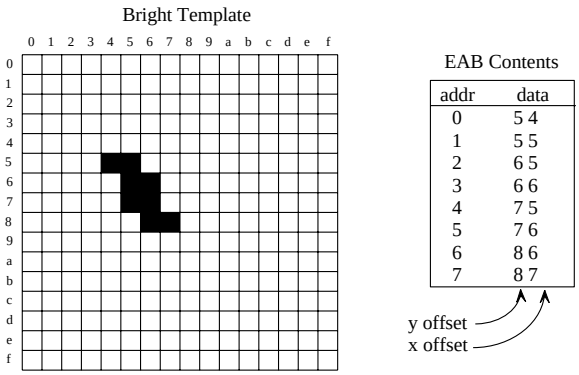


Figure 4.1: Example of Bright Template Storage

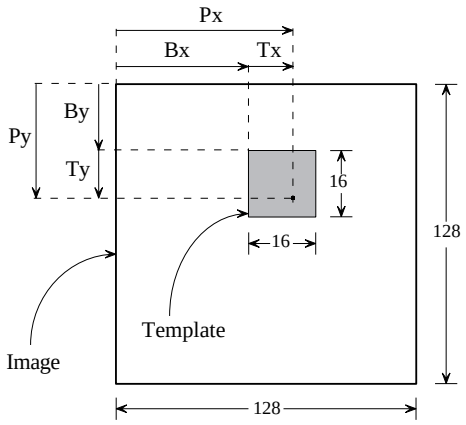
to the image pixel that needs to be accumulated for the shapesum. This is depicted in Figure 4.2.

With the bright template information stored in this format, a pixel that needs to be accumulated can be fetched every cycle. Each cycle a new offset is read from the EAB, it is added to the base image offset, and the result is used to address the off-chip RAM. The pixels that do not lie under on-bits are not needed in the calculation and so are not accessed. The increase in performance from this method is due to the sparseness of the bright templates. Since an average template has no more than ten percent of its bits on, this method takes one tenth or fewer of the cycles of a straightforward correlation that accesses all the pixels in the template region.

Figure 4.3 shows the basic layout of the shapesum calculation. An offset to a template on-bit is read from the EAB, the offset is added to the current image offset (x offsets are added together as are y offsets), and this sum forms the address to the image pixel that needs to be accumulated. The pixel is read from the image RAM and added to the running total. When all the template on-bits have been processed, the shapesum is complete for that image position.

4.2 Threshold Implementation

The threshold calculation shown in Figure 4.4 takes the result of the shapesum calculation, divides it by the number of on-bits in the bright template, and subtracts a



Bx = Base Image x Offset
By = Base Image y Offset

Tx = Template x Offset
Ty = Template y Offset

Px = Pixel x Offset = Bx + Tx
Py = Pixel y Offset = By + Ty

Figure 4.2: Pixel Address Calculation

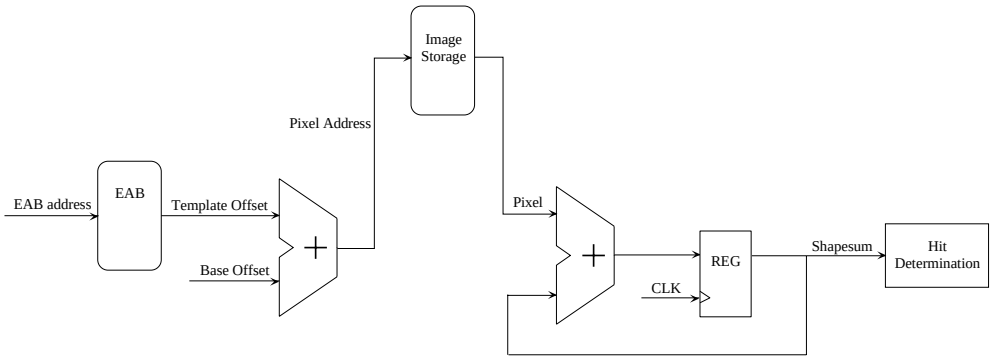


Figure 4.3: Shapesum Calculation for Single Template

constant from it. The constant is loaded into the FPGA and stored in a register at the

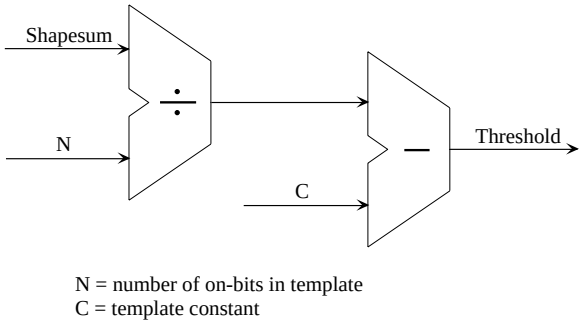


Figure 4.4: Threshold Calculation

same time the EAB is loaded with the template data. The loading of the template data and constants occurs during configuration. The resulting threshold is stored in a register so that it can be used by the brightsum and surroundsum operations.

The divider is implemented as an iterative divider [6]. An iterative divider in general takes as many cycles as the number of bits in the dividend. The dividend in this case is the shapesum. The number of bits in the shapesum varies with the number of on-bits in the bright template and the value of the pixels in the image. To be safe, a width of 14 bits is assumed. This allows 64 on-bits in the template with each pixel having a value of 255, the maximum it can be. Fourteen bits is conservative since most templates will have far fewer than 64 on-bits, and the pixels will have lower values than 255. An iterative divider was chosen to conserve space as it is very compact. The number of cycles required to perform the division does have an impact on performance. Performance is discussed in Chapter 7.

A different type of divider that requires fewer cycles could be beneficial. One option that was explored was to multiply by the reciprocal of N, the number of on-bits, instead of dividing by N. Since N is a constant, a very compact, fast multiplier can be used [7]. Such a multiplier was designed and tested for this project. This multiplier was approximately 1.6 times the size of the iterative multiplier but required only three cycles instead of 14. That is 4.7 times faster when measured by number of clock cycles. The difficulty that was encountered in multiplying by the reciprocal was achieving the

required precision in the result. Due to rounding effects, the result can be off by ± 1. Another drawback to using a constant multiplier is that it would require reconfiguration of the FPGA between templates. This is because the constant is hard-coded into the circuit so the multiplier becomes template-specific. Since the iterative divider is more general, full reconfiguration is not necessary. All that is required to change templates is to reload the EABs and a few registers. The reconfiguration time for the FPGA is several orders of magnitude greater than the time required to reload the EABs. Section 7.1.5 and Table 7.1 on page 42 compare the performance of systems using each of the two methods.

4.3 Brightsum Implementation

The brightsum calculation is similar to the shapesum calculation. The image pixels that lie under the on-bits in the bright template are used in the brightsum just as they are in the shapesum. To access these pixels, there are at least two possibilities: 1) Use the same kind of address generation that is used for the shapesum and read the pixels from an off-chip RAM, or 2) Fetch the pixels once for the shapesum and reuse them for the brightsum. The second option is the one that was implemented. This method will be referred to as pixel caching.

4.3.1 Pixel Caching

Since the brightsum uses exactly the same pixels that the shapesum uses, image memory can be conserved by fetching the pixels just once and using them for both the shapesum and the brightsum. The only difficulty is that the brightsum operation must wait for the threshold operation to complete before it can begin. To reuse the pixels, they must be stored on the FPGA when they are fetched for the shapesum so they can be used later when the brightsum begins. The Altera EABs are a good place to store these pixels.

The brightsum operation uses the threshold result in its computation. This dependency prevents it from running concurrently with the threshold operation at the same image position. It does not, however, prevent the two operations from running simultaneously at different offsets. It is advantageous, then, to run the threshold and

the brightsum simultaneously with the threshold operating at one image position ahead of the brightsum. When the threshold is finished being calculated, it can be stored in a register so that it is available for the brightsum calculation. That is how the system was implemented.

When the pixels are fetched for the shapesum operation, they are stored temporarily in an EAB. After the threshold is complete for a given offset, the brightsum operation at that offset begins and uses the cached pixels instead of going off chip to fetch them. Going off chip would require an extra image RAM for the brightsum plus I/O pins to access it. It would also require an address generation circuit similar to that of the shapesum.

Caching the pixels temporarily on chip reduces the image memory and FPGA pin requirements. It also eliminates the need for address generation in the brightsum calculation. It isn't free, however. It does require on-chip RAM to store the pixels. Not only that, since the threshold operation is storing pixels at the same time the brightsum is reading them, the on-chip RAM must be able to be written and read in the same cycle. The FLEX 10K EABs are not capable of doing this. They can be written and read, but the value read is the value that is being written. What is needed is to read the previous value and write a new one in the same cycle. Otherwise two cycles are required which doubles the cycle count of the shapesum and brightsum. The way around this is to use two EABs. The shapesum stores pixels to one EAB while the brightsum reads pixels from the other. After the operations finish, they swap roles so that the brightsum gets the pixels that were just used for the shapesum. With the pixels cached, the rest of the brightsum calculation is simple: compare each of the cached pixels against the threshold that was calculated previously and keep a count of how many pixels are greater than the threshold.

4.4 Surroundsum Implementation

The surround templates do not have the same sparseness that the bright templates have; this makes the optimization that was used for the brightsum less effective for the surroundsum. It becomes even more so in the parallel implementation discussed in Chapter 5. In view of this, the surroundsum operation was not implemented in the same way as the brightsum operation. Instead, the surround template was divided

into four regions and all four regions are calculated in parallel. Although this approach quadruples the amount of hardware required to compute the surroundsum, it leads to a more balanced implementation where the time to compute the shapesum/brightsum is nearly the same as the time required to compute the surroundsum. As these operations operate in parallel, it is important that they proceed at the same rate.

The implementation uses an EAB to store the template data. One bit is stored for each location in the template. As the pixels are read from the off-chip RAM, they are compared to the threshold. If the pixel value is less than the threshold, and if the template bit is a ‘‘1’’, a counter is incremented.

The hardware requirements for each quadrant processor for the surroundsum are a counter, a comparator, an off-chip RAM for image storage, and FPGA pins to access the RAM. This is shown in Figure 4.5.

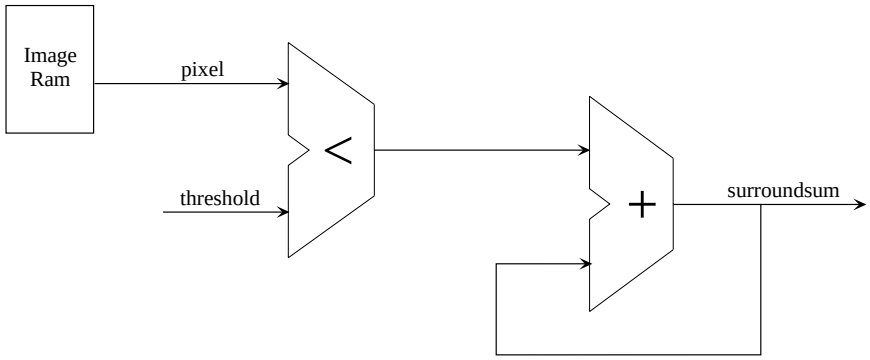


Figure 4.5: Surroundsum Layout

4.5 Hit Determination Implementation

The implementation of the hit determination is straightforward. The results from the threshold, brightsum, and surroundsum are compared to the appropriate constants (see Section 2.2.5, p. 11) and a hit is scored if they pass the comparisons.

4.6 Microcode Method

Storing offsets as described in this chapter will be referred to as the ‘‘microcode’’ method. This is because, in essence, the template offsets are microinstructions that are loaded into the FPGA RAM and then used to control the operation of the processor. This satisfies the goal of using the embedded RAM for circuit control. On the other hand, the template offsets could be considered data that is loaded onto the FPGA periodically and used in the address calculation. This shows that the distinction between data and instructions is not always clear.

Chapter 5

MULTI-CHUNK PROCESSOR

Chunky SLD is inherently a parallel algorithm. Multiple chunks can be processed simultaneously since there is no data dependency between chunks. The degree of parallelism that can be achieved is limited only by the hardware available, not by the data or by the chunky SLD operation itself. This chapter describes how the chunky SLD implementation that was explained in the previous chapter can be parallelized. The parallel version is the one actually implemented for this project.

5.1 Sharing Memory between Chunk Processors

Making the implementation parallel is not simply a matter of replicating the single chunk processor multiple times across the FPGA. If this were done, very few processors would fit on a device due to the memory requirements of each processor. In the implementation outlined in the previous chapters, the memory required to process a chunk consists of the following:

- 5 off-chip image memories,
- 1 EAB for bright template storage,
- 1 EAB for surround template storage, and
- 2 EABs for image pixel caching.

The five off-chip memories include four for the four quadrants of the surroundsum and one for the shapesum. Recall that the brightsum does not need an off-chip image RAM because of pixel caching. Two EABs are required for pixel caching because the EAB does not have the necessary read and write capability.

The memory requirements listed above seriously limit the number of processors that can fit on an FPGA. The largest Altera device at the time of this writing is the 10K100 which has 12 EABs. This means a maximum of three processors with the

above memory requirements can fit on the 10K100. Three processors would severely under-utilize the logic resources on the 10K100. Thus it is necessary to share the memory between chunk processors in some way so that a greater number of processors fit on a device. The following sections describe how this is done.

5.1.1 Shapesum

The memory requirements for the shapesum include the EAB for bright template storage and the off-chip RAM for image storage. To share the EAB among processors, the bright templates need to be combined in some way so that they do not require one EAB for each processor.

As discussed earlier, the bright template is stored in the form of offsets to the on-bits. These are the offsets that are added to a base offset to calculate the address of a pixel that needs to be accumulated. To reduce the on-chip RAM requirements for this bright template storage, several templates may be combined so that they use one EAB between them to store their offsets.

This is done by first selecting a group of templates that will be run in parallel. From this group, a master template is created. The master template contains the on-bits from each of the member templates, as shown in Figure 5.1. This is essentially an OR-type operation between the templates. In other words, if the templates are viewed as sets of bits, the master template is the union of the templates in the group. The master template is then stored as a normal template would be stored in the EAB--the offsets to the on-bits are stored. The master template information is used in the same way as before, that is, each offset is added in turn to the base image offset to obtain the address of the next pixel to be fetched. Now, however, only one EAB is required to store the template for the whole group.

Template Masks

The obvious problem is, not every pixel fetched is valid for every template in the group. Because the master template contains the union of the group of templates, many of the pixels that are brought onto the FPGA are destined for only some subset of the templates in the group. Some way is needed to distinguish which pixel goes to which

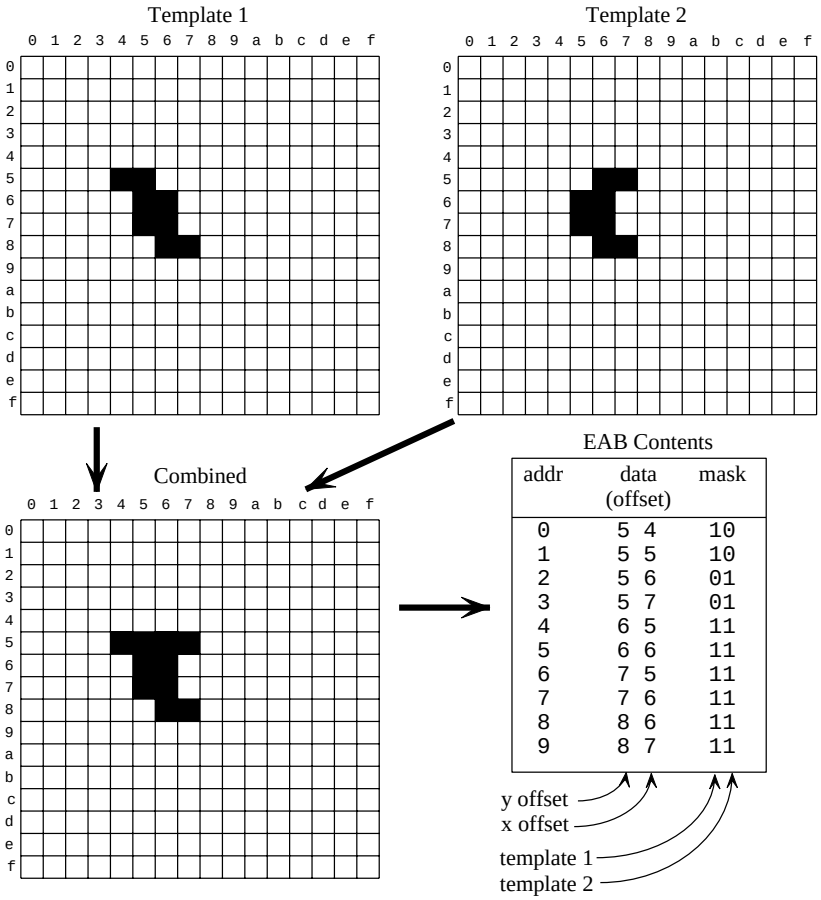


Figure 5.1: Example of Combined Template Storage

template processor. To accomplish this, another on-chip RAM is used to store a mask for each of the master template offsets. This is the mask data shown in Figure 5.1. There is a mask for each offset in the master template. Each mask has one bit per template. The mask bit indicates whether the offset is valid for the template which it represents, and it is used to control the accumulator for the shapesum calculation for that template.

Figure 5.2 shows an example of the shapesum processor for two combined templates. One pixel is retrieved from the off-chip image RAM and is fed to the shapesum calculators for both templates. At the same time the appropriate mask is read from an EAB and each bit is sent to the shapesum processor to which it corresponds. The mask bit controls the accumulator, determining whether or not to add the incoming pixel to the current accumulated value.

Using this method of combining bright templates for the shapesum, only two on-chip RAMs are needed to store the template information for the whole group of bright templates. Also, only one off-chip RAM is needed for image storage for the group of templates. Less off-chip RAM also means that fewer FPGA pins are required to access it.

5.1.2 Brightsum

The memory requirements for the brightsum processor include two EABs for pixel caching. No off-chip RAM is needed.

Combining Pixel Caching

The mask information that is used in the shapesum operation can also be used to combine the image caches for all the templates in the group. Each pixel for the master template is cached in an on-chip RAM as it is fetched for the threshold calculation. Then, when the pixels are needed for the brightsum calculation, they are read from the RAM and broadcast to the brightsum calculators for all the templates. If the pixel is not valid for a particular template, the mask bit prevents the counter from incrementing.

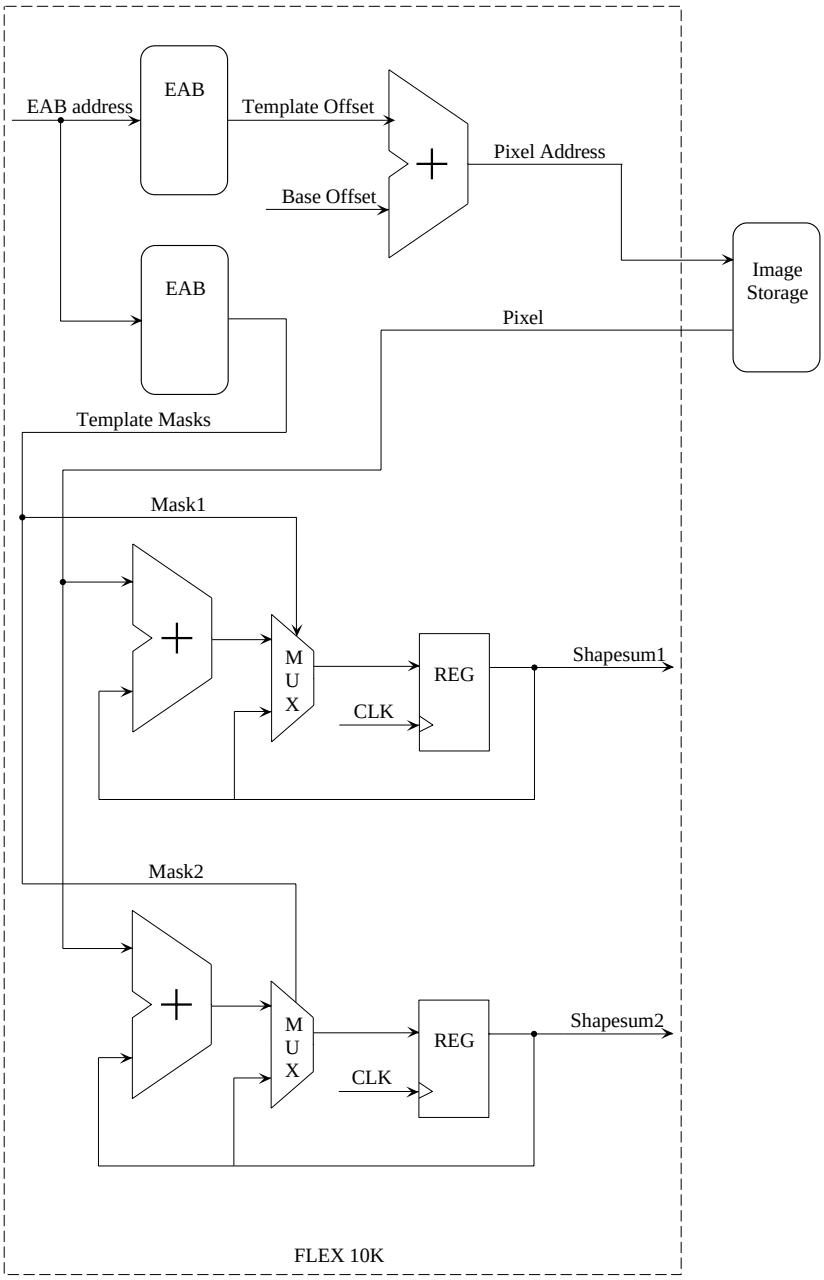


Figure 5.2: Shapesum Calculation for Combined Templates

Hardware Savings

When the cache EABs are combined like this, the whole group of combined templates uses only two EABs total instead of two EABs for each bright template.

5.1.3 Surroundsum

The surroundsum operation is already set up to share memory resources between template processors. It reads every pixel in the template region and uses mask bits to determine whether to increment its counter. All that is needed to process more templates is to add a mask bit for each template in the group for each template bit location.

5.1.4 Impact of Sharing Memory

To illustrate the savings that sharing memory produces, Table 5.1 compares the RAM requirements for a parallel implementation with and without memory sharing.

Table 5.1: RAM for 8 Chunk Processors

		Shared	Not Shared
Off-Chip RAM	Shapesum Image	1	8
	Surroundsum Image	4	4
	Total	**5**	**12**
On-Chip RAM	Bright Template Offsets	1	8
	Bright Mask	1	0
	Surround Mask	4	4
	Pixel Cache	2	16
	Total	**8**	**40**

It shows the RAM required for a group of eight templates using combined memory resources versus the same group of eight using individual resources. This table can be compared to Table 3.1 on page 14 which lists the number of EABs available on each member of the 10K family. With the combined template method, one 10K40 has enough on-chip RAM for eight chunk processors. Without combining templates, the same eight chunk processors would require five 10K40s to satisfy the on-chip RAM requirement.

Memory requirements do not tell the whole story. Sharing memory has the potential to impact performance negatively, particularly in cycle count. The reason for this is that combining bright templates may produce a master template with a large number of on-bits. A large number of on-bits results in a high number of required memory reads. Chapters 6 and 7 discuss this issue more completely.

5.2 Final Implementation

There really is no ‘‘final’’ implementation because the system is quite flexible. Different numbers of processors may be grouped together and different sized devices can be used. With that in mind, though, a set configuration was chosen so that some performance results could be generated.

The configuration that was used consists of eight chunk processors. The device chosen was a FLEX 10K50. Table 5.1 shows that a group of eight chunk processors requires eight EABs. The 10K50 has 10 EABs available. It also has enough logic and routing resources for eight processors.

Figure 5.3 shows a basic block diagram of the eight processor system. Only two processors are shown with ‘‘...’’ representing the other six.

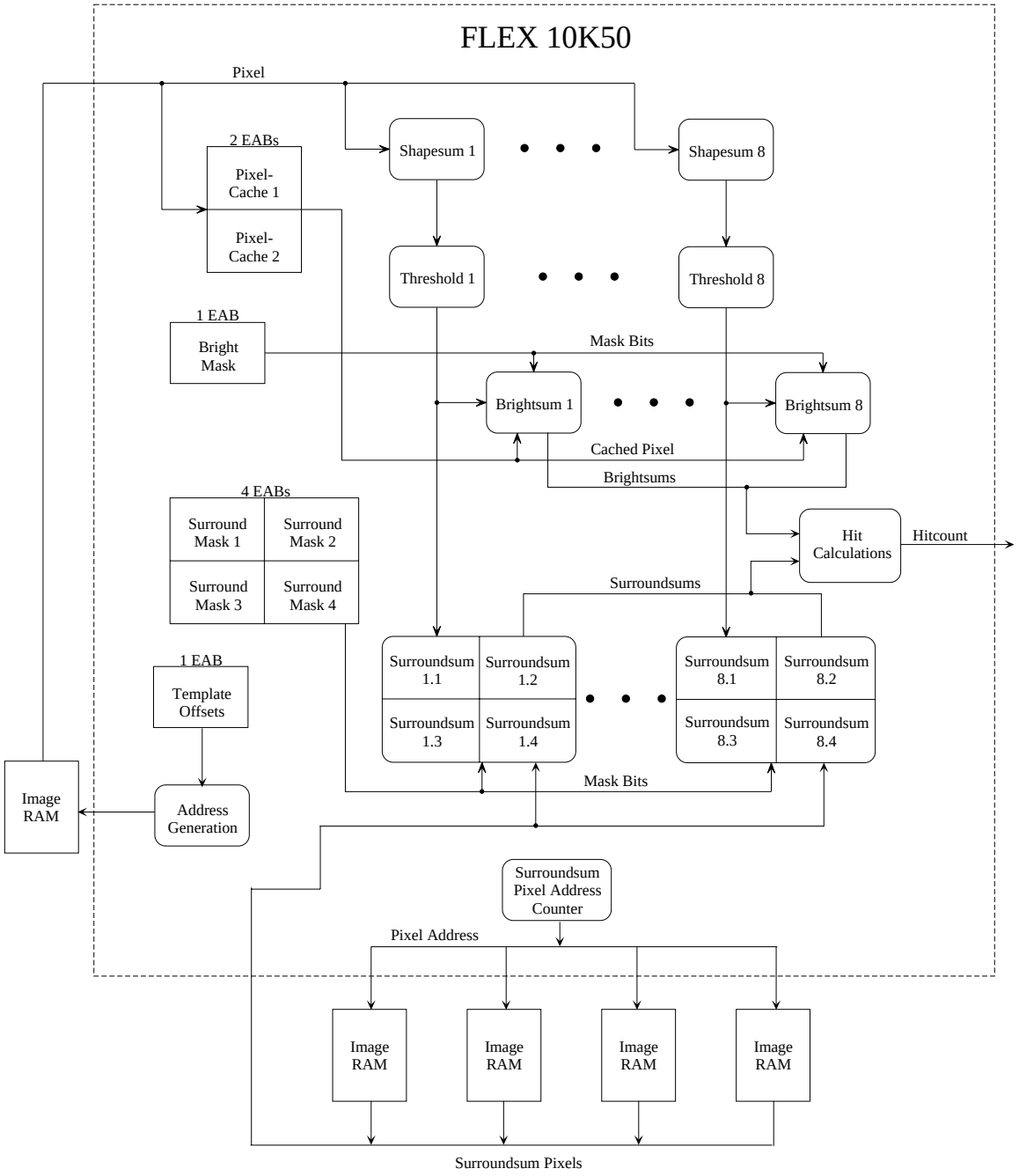


Figure 5.3: Block Diagram of 8 Processor System

Chapter 6

TEMPLATE SYNTHESIS AND ANALYSIS

The performance of the parallel implementation (Chapter 5) is dependent on the bright template data because of the method used to combine templates. Due to the obvious need to quantify performance for this system, it became necessary to analyze template data to make a performance prediction. Real templates, however, were not available to analyze. Sandia Laboratory provided 17 bright and 17 surround templates that they said were representative of actual templates. These were studied, but synthetic bright templates were also generated for further analysis. This chapter discusses the results. Surround template data does not affect performance so no surround template data was generated.

6.1 Template Synthesis

A template model and synthesis algorithm were used to generate the synthetic templates. The model was chosen based on the characteristics of the bright templates. The synthesis algorithm was chosen based on its ability to generate data based on the model.

6.1.1 Template Characteristics

The most important template characteristics are the number of on-bits and their spatial relationship. The sample templates from Sandia and other available information implied that the bright templates are:

- sparse,
- clustered, and
- centered.

Sparse: This detail was given explicitly by Sandia. What it means is that there are few on-pixels or ‘1’s compared to the total number of pixels in the template.

The figure given by Sandia was that bright templates are less than ten percent populated. The total number of bits in a template is 256. Ten percent of this is 26 bits.

Clustered: This means that the on-bits are generally in groups and not scattered across the template. This is an assumption based on how the templates are formed and what they represent. To form a bright template, a SAR (Synthetic Aperture RADAR) image of a target is thresholded to make it binary. It is then divided into 40 pieces. Each piece becomes a template. This is done in an attempt to identify pieces of the target so that partially obscured targets can be recognized. Since each bright template identifies a piece of a target, it seems likely that the on-bits will be clustered in groups.

Centered: This means that the cluster or clusters of on-bits are in the center of the template, and very few pixels, if any, are on the edges of the templates. This is probable because the result of the algorithm does not depend on the position of the on-bits within the template. The center of the template is the most likely place for the on-bits to be positioned.

The 17 templates provided by Sandia generally exhibited these characteristics. Each template had eight on-bits that were clustered in the center of the template. Figure 6.1 shows eight of the sample templates.

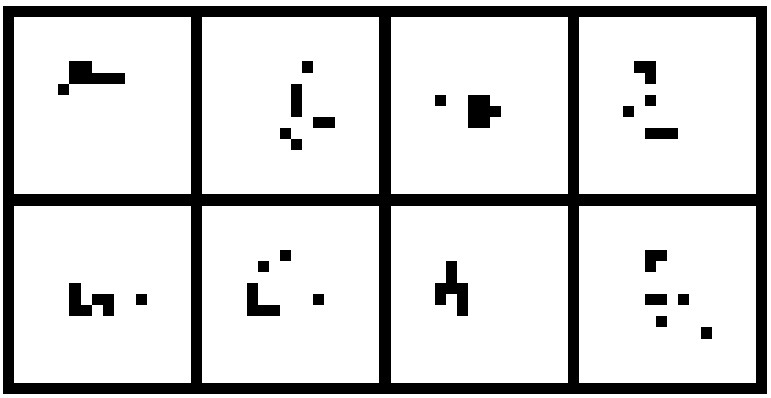


Figure 6.1: Sample Templates from Sandia

6.1.2 Template Model and Synthesis Algorithm

A binary Markov Random Field (MRF) [8] was chosen as the model for the bright templates. A MRF is a statistical model for two dimensional data. It was chosen for its ability to model pixel clustering in images. The MRF did not have the ability to model the other template characteristics, but the synthesis algorithm provided the means to incorporate these. The synthesis algorithm was taken from [9]. The model and algorithm have parameters α and β. The exact meaning of these parameters is not important here. Appendix A may be consulted for further details. It is enough to say these parameters control the strength of the clustering in the generated templates. The number of on-bits can also be controlled by the algorithm.

6.2 Analysis of Synthesized Templates

Many templates were synthesized to analyze the effects of varying the α and β MRF parameters. By varying these parameters, the degree of clustering was varied in the generated templates. The number of on-bits per individual template was also varied. These two factors, the clustering strength and the number of on-bits per template, are what determine how well the templates combine. The intent of the synthesis was to see how much clustering and what maximum number of on-bits per template are required for good performance. After the templates were generated, groups of eight and sixteen templates with the same parameters were combined and a master template generated to see how many on-bits resulted in the combined group. For best performance, the number of on-bits in the master template should be 50 or fewer (see Section 7.1.1). This must be possible, according to the Sandia information, at up to 25 on-bits per template.

The results showed that the ideal case is unlikely. That is, it is unlikely that the master template will have fewer than 50 on-bits when the member templates each have 25 on-bits. In fact, none of the parameters that were synthesized resulted in fewer than 50 master template on-bits with 25 on-bits per individual template. The closest were groups similar to the one shown in Figure 6.2, and templates with this high degree of clustering do not seem very likely. This group has parameters $\alpha = -4.0$, $\beta_{1,*} = 2.0$, and $\beta_{2,*} = 3.0$. Up to 22 on-bits per template were possible before the resulting master template had more than 50 on-bits.

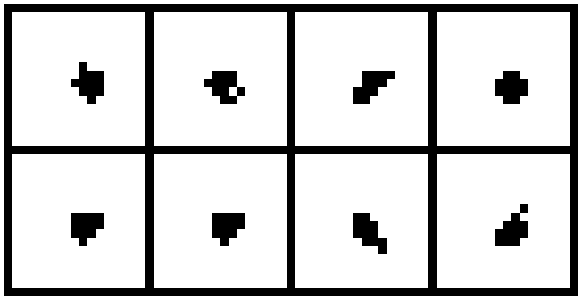


Figure 6.2: Highly Clustered Synthesized Templates

On the other hand, the analysis did indicate that the number of on-bits in the master template is likely to be within reasonable limits for good performance. To make a realistic performance prediction, it was necessary to estimate the clustering parameters of real templates by using the templates that were provided by Sandia. Figure 6.3 shows synthesized templates with 12 on-bits each that seem to resemble the degree of clustering in the Sandia templates. Table 6.1 shows the parameters that correspond to the synthesized templates in the figures. This table also shows two other figures of interest called the ‘‘Ideal Case’’ and the ‘‘Worst Case’’. The ideal case is the maximum number of on-bits in each member template that is allowed for the master template to have fewer than 50 on-bits. In other words, this is the maximum number of on-bits per template that is allowed for maximum performance. The worst case table entry is the number of master template on-bits for groups with 25 on-bits per template. If the bright templates are truly ten percent populated, the worst case number shows how many on-bits the master template will have.

Table 6.1: Estimated Parameters of Sandia Sample Templates

Group	$\beta_{2,*}$	α	$\beta_{1,*}$	Ideal Case	Worst Case
0	0.0	-5.6	2.8	9	90
1	1.0	-4.0	2.0	9	70
2	2.0	-2.4	1.2	20	55
3	3.0	-1.6	0.8	23	52

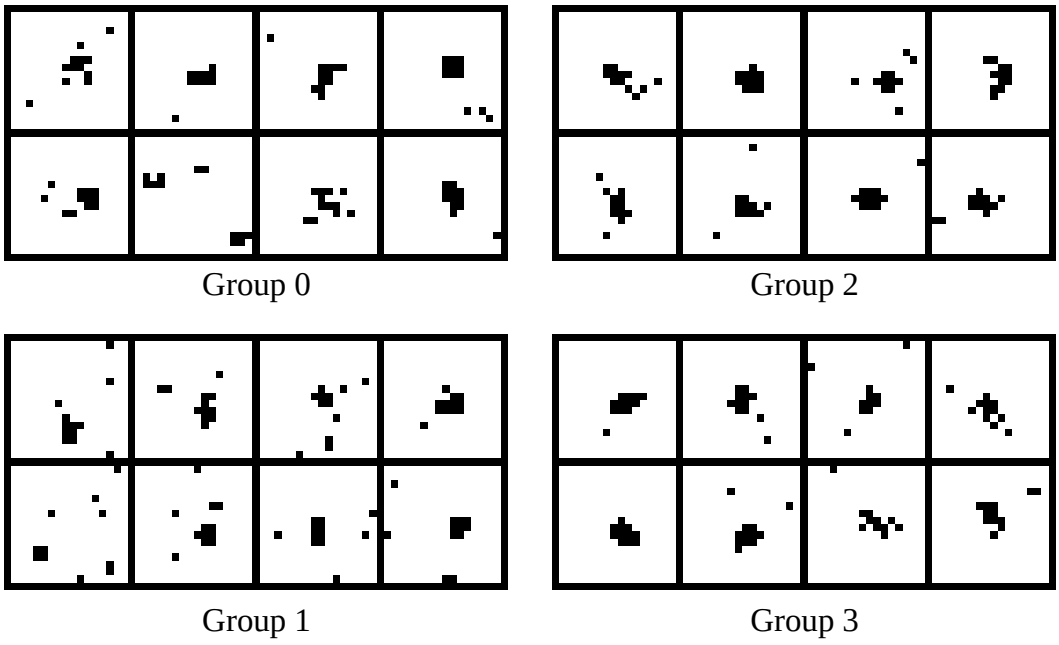


Figure 6.3: Synthesized Templates that Resemble Sandia Sample Templates

The analysis shows how well typical templates might be expected to combine. Assuming the templates in Figure 6.3 are indeed typical, the worst that is likely is that the master template will have somewhere around 100 on-bits. The best that can be expected is that it will have fewer than 50 on-bits with the individual templates being almost ten percent populated. The next chapter elaborates what this means for performance.

Chapter 7

RESULTS AND PERFORMANCE

The design for this project was done completely in VHDL. The VHDL was synthesized to produce a circuit that can be down loaded to an Altera FLEX 10K50. The necessary hardware was not available, so the design was never tested on a real device. The performance results come from the software tools.

Two major CAD software tools were used. Synopsys was used to synthesize the VHDL source code and output an EDIF file. The EDIF file was read by MAX+PLUS II which is Altera's software. MAX+PLUS II placed and routed the design and prepared a bitstream ready to be down loaded. This chapter discusses the performance of the system using the figures from these CAD tools. Several CAD tool issues are also discussed.

7.1 Performance

The performance of this system is the product of several factors. The principal contributing factors are:

- number of clock cycles to complete one iteration,
- clock frequency,
- template loading time, and
- FPGA area required for a chunk processor.

These factors figure into the performance in the following ways. The number of clock cycles along with the clock frequency dictate the time of execution for one iteration. An iteration is the calculation of one chunk at one image position. The template loading time affects how fast the FPGA can be configured to process a new template. The required circuit area actually affects performance per device, rather than raw performance. The

size of the circuit limits how many chunk processors fit on one FPGA and therefore how many chunks can be processed in parallel with one device.

To see what kind of performance can be achieved, performance factors are discussed individually. The implementation used to measure the performance is the one presented in Section 5.2. This is a system with eight chunk processors on a FLEX 10K50.

7.1.1 Clock Cycles

The number of clock cycles required to complete a chunk calculation is dependent both on the implementation and on the device used for the implementation. The number of cycles is given by

$$n_{cycles} = \max(n_{thresh}, n_{bright}, n_{surround}) \tag{7.1}$$

where n_{thresh}, n_{bright}, and $n_{surround}$ are the number of cycles to calculate the threshold, brightsum, and surroundsum, respectively. Because all three operations run simultaneously, the one that requires the most cycles determines the number of cycles for the whole chunk calculation.

Immediately it is possible to establish a lower bound on n_{cycles} in the above equation. Since $n_{surround}$ is fixed, it represents the fewest number of cycles in which one chunk can be processed. The reason that $n_{surround}$ is fixed is that the surroundsum processor must read and process every pixel under the template. It does this four pixels at a time so 64 cycles are required. After the four quadrants are processed, several cycles are required to combine the results. To cover this and other overhead, 70 will be used as the figure for $n_{surround}$. That means that the minimum that n_{cycles} can be is 70.

The brightsum and threshold do not have a fixed number of cycles. They depend on the number of on-bits in the bright template. The one thing that can be said for certain is that the threshold always uses more cycles than the brightsum for a given template. This is because the threshold includes the shapesum, and the shapesum requires roughly the same number of cycles as the brightsum. After the shapesum completes, the division is performed requiring an additional 14 cycles (see Section 4.2).

These observations make it possible to rewrite 7.1 as

$$n_{cycles} = \max(n_{thresh}, 70) \tag{7.2}$$

This makes it clear that the number of cycles required for the threshold is really the key to calculating the number of cycles for the whole algorithm. So, just how many cycles are required for the threshold? As just mentioned, the divide requires 14 cycles after the shapesum completes. That is the starting point in the calculation. Then, each on-bit in the master template requires a memory access and each memory access requires one cycle. This means that the number of on-bits in the master template plays an important role in the performance. This is the reason the results of Chapter 6 are important. The analysis there shows how many on-bits can be expected from templates with varying degrees of clustering.

The ideal case occurs when the number of on-bits for the master template is below about 50. If this is the case, then the cycles for memory reads plus the divide cycles is fewer than the 70 cycles for the surroundsum. Equation 7.2 shows that this results in the fewest cycles possible. If the number of on-bits in the master template is greater than 50, every on-bit above 50 adds a clock cycle to the total number required. For example, if the number of on-bits is 100, the number of cycles required is roughly $100 + 14 = 114$.

7.1.2 Clock Frequency

Significant effort went into achieving the best clock frequency possible, both in the circuit design and in the use of the software tools. In the circuit design, pipelining was used wherever possible to reduce the length of combinational logic paths. Specific VHDL coding techniques were employed that help the synthesis tools to obtain a short clock period. Various CAD tool options were tried to find the best combination (see Section 7.2). With this effort, a frequency of 25 MHz was achieved, as reported by the place and route tool. This was a significant improvement over the 12 MHz obtained on the first synthesis attempt.

7.1.3 Template Loading Time

When a chunk processor must process a new template, it loads several pieces of data. These include the microcode template offsets, the template masks, and the various constants involved in the computations. A simple protocol for loading this data

was designed and implemented. This process will be referred to as reconfiguration despite the fact that the FPGA is not actually reconfigured in the traditional sense. This reconfiguration requires approximately 500 clock cycles. The clock frequency is the same as the execution clock frequency, or 25 MHz. The time to load new template data is calculated by multiplying the number of cycles by the clock period. This gives $500 \times 40ns = 20\mu s$.

The reconfiguration time would be substantially worse if the entire FPGA had to be reconfigured. This would be necessary, for example, if the divider were implemented as a constant multiplier (Section 4.2). The published configuration time for the Altera 10K50 is $70ms$. There is some possibility that a faster configuration mode may become available for the 10K parts which would be 40 times faster than the current mode. That would make the reconfiguration time approximately $1.8ms$.

7.1.4 FPGA Area

The area required for the design is 2300 logic elements. Each logic element (LE) contains one 4-input lookup table and one flip-flop. The 10K50 has 2880 LEs. This gives a utilization of 80%. The remainder of the LEs were unusable because of insufficient routing resources. In fact, hand placement was required to get the design to route even at 80% utilization.

7.1.5 Overall Performance

To combine the individual performance factors into one overall figure, a performance metric was needed. One metric that was used is the time required to process an entire Q of 2880 chunks. This allows the inclusion of execution time as well as reconfiguration time in the performance figure. The calculation was made for the system described in Section 8.2 which processes 40 chunks in parallel using five Altera 10K50s. The necessary parameters to calculate the execution time are:

- $n_{cycles} = 70 =$ number of cycles to calculate one chunk at one offset. As already noted, this may actually be more than 70, but 70 is good for a first estimation.
- $n_{offsets} = 65^2 =$ number of offsets per image. This is the size of the search region given in Section 2.1.1.

- $n_{orients} = 72 =$ number of orientations for each class.
- $t_{cycle} = 40ns =$ clock period.

The reconfiguration time can be calculated with the following parameters:

- $t_{onereconfig} = 20\mu s =$ time for a single reconfiguration.
- $n_{reconfig} = 72 =$ number of reconfigurations that must be done for each Q. Since 40 chunks are processed in parallel, reconfiguration takes place one time per orientation.

The total time, t_Q, required to process one Q is the sum of the execution and reconfiguration times, or

$$t_Q = t_{exec} + t_{reconfig}, \tag{7.3}$$

where t_{exec} is the execution time and $t_{reconfig}$ is the time to load new template data. t_{exec} is given by

$$t_{exec} = n_{cycles} \times t_{cycle} \times n_{offsets} \times n_{orients}. \tag{7.4}$$

$t_{reconfig}$ is given by

$$t_{reconfig} = t_{onereconfig} \times n_{reconfig}. \tag{7.5}$$

Substituting values into 7.4, 7.5, and 7.3 gives

$$\begin{aligned} t_{exec} &= 70 \times 40ns \times 65^2 \times 72 && (7.6)\\ &= 851ms, && (7.7)\\ t_{reconfig} &= 20\mu s \times 72 && (7.8)\\ &= 1.4ms, && (7.9)\\ t_Q &= 851ms + 1.4ms && (7.10)\\ &= 852ms. && (7.11) \end{aligned}$$

To summarize the above equations, the system in Section 8.2 consisting of five Altera FLEX 10K50 FPGAs and nine image RAMs can process one Q in $852ms$. This figure

includes reconfiguration time. The calculations do not include time to tally the hitcounts from the individual FPGAs. That circuitry has not been implemented. If the templates do not combine well so that more than 70 cycles/offset are required, it will take longer to process a Q. For example, If 120 cycles are required, the time for one Q can be calculated by

$$t_Q = \frac{120}{70} \times 852ms \tag{7.12}$$

$$= 1.46s. \tag{7.13}$$

The reconfiguration time is only 0.2 percent of execution time which is insignificant. This is because the FPGA does not actually have to be reconfigured with a new bitstream. If the FPGA did have to be completely reconfigured, the total reconfiguration time for a Q would go from $1.4ms$ to $4.9s$, using the published configuration time for the 10K50. If the fast mode reconfiguration is used, the total reconfiguration time becomes $123ms$. A system using a constant multiplier instead of the iterative divider would have to reconfigure completely but would use fewer cycles for the division. Table 7.1 compares the performance of systems using the iterative divider and the constant multiplier. Because of the increased reconfiguration time for the constant multiplier, essentially no performance is gained from using it even though it is faster.

Table 7.1: Comparison of Systems Using Iterative Divider and Constant Multiplier

Division Type	Cycles/Offset	Execution Time	Reconfiguration Time	Time/Q
Iterative	120	$1.46s$	$1.4ms$	$1.46s$
Constant	109	$1.33s$	$0.123s$	$1.45s$
Constant	109	$1.33s$	$4.9s$	$6.2s$

Another closely related method of expressing performance is to calculate how many Q per second can be processed. For the example with 70 cycles/offset, the performance measured this way is $1/852ms = 1.2\ Q/s$. Of course, this is with five FPGAs. To make performance comparisons with other systems, it is helpful to measure

this performance per FPGA. In that case, the Q per second figure should be divided by five (the number of FPGAs in the proposed system). Continuing the example,

$$\frac{1.2\ Q/s}{5\ FPGA} = 0.235\ Q/s/FPGA, \tag{7.14}$$

that is, 0.235 Q per second per FPGA, assuming 70 cycles per offset. Table 7.2 summarizes the above performance figures for 70 and 120 cycles per offset.

Table 7.2: Performance of the Altera Implementation

Cycles/Offset	Time/Q for System	Q/s for 1 FPGA
70	0.852 s	0.235
120	1.46 s	0.137

7.1.6 Performance Comparison

To better understand what these performance figures mean, it is helpful to compare them to the performance of other implementations. Several implementations other than this one have been studied at Brigham Young University. A comparison will be made with two other systems: a Xilinx 6200 implementation and a SPLASH-2 [10] implementation [11]. The performance of a workstation is also presented as a baseline. Performance for the comparison is measured in the number of Q that can be processed per second (Q/s). Since the three systems all use multiple FPGAs, the figure for each system will be normalized by the number of FPGAs in that system. The resulting figure is then on a per-device basis. The following sections describe briefly how the performance for each system is calculated. Following that, Table 7.3 summarizes the performance comparison.

Altera

For the Altera implementation, the appropriate performance figures are shown in Table 7.2 in the last column.

Xilinx 6200

The Xilinx 6200 system is currently under development so the performance figures are projected. The implementation uses a bit-serial approach on two Xilinx 6216 parts. The projected clock speed is 50 MHz with 52,540 cycles required to calculate one chunk over a 65×65 search region. The reconfiguration time between templates is equal to about 10 percent of the execution time. Putting these figures together gives 0.157 Q/s for the 6200 implementation.

Splash-2

The Splash-2 approach, which is detailed in [1] and [11], was developed using the Xilinx 4010 FPGAs of Splash-2. These parts are relatively old. The performance for a system using the newer 4010E series was also projected in [1]. To make the comparison more realistic, the 4010E figures are used for the comparison. They predict that a two-board Splash system with 32 4010E parts can process 40 chunks over a 128×128 region in 26 ms. The calculations for the Altera and Xilinx 6200 system used a search region of 65×65 which is one fourth the number of pixels of a 128×128 system. Consequently, the Splash-2 figure will be reduced to one fourth of 26 ms or 6.5 ms. Making the calculations for an entire Q and normalizing by the number of FPGAs required gives 0.0668 Q/s for the Splash-2 implementation.

Workstation

Rencher compared the Splash-2 implementation to the performance of a general purpose workstation in [11]. He reports that an HP 770 workstation running at 110 MHz required 59 seconds to process one orientation. By extrapolation this means it would require 4248 seconds to process an entire Q. This calculation was done using a 113×113 search region. If the calculation time is reduced to compensate for the 65×65 search region used in the other calculations, the workstation performance is 711×10^{-6} Q/s.

Table 7.3 summarizes the above figures and gives the speedup factor over the workstation implementation. The table shows that the Altera and the 6200 implementations are in the same range of performance. The Splash-2 implementation

is somewhat slower than the other two. This can be attributed to a slower clock speed (22.2 MHz) as compared to the Xilinx 6200 implementation, and the sparse template exploitation of the Altera implementation. Also, the size of the device is not figured in. For example, the Xilinx 4010 parts of Splash-2 have 800 LUTs each while the Altera FLEX 10K50 has 2880. That is a factor of 3.6 difference. If the Q/s figure for Splash-2 is multiplied by 3.6 to compensate, the result is 240 Q/s. The Xilinx 6200 uses a fine-grained architecture which makes it more difficult to compare its size to that of the other two. There are yet other factors such as cost of the system and power consumption that are not included in the comparison. Given these shortcomings, the figures in the table should only be taken as a starting point for a performance comparison.

Table 7.3: Performance Comparison

Platform	cycles/offset	Q/s	Speedup
HP 770	-	711×10^{-6}	1.0
Altera	70	235×10^{-3}	330
Xilinx 6200	-	157×10^{-3}	221
Altera	120	137×10^{-3}	193
Splash-2	-	66.8×10^{-3}	94

7.2 CAD Tools

The CAD tools greatly influence the performance of a VHDL-designed circuit such as this one. The tools perform various optimizations in an attempt to achieve the best speed and area results.

7.2.1 Synopsys

There are many options available in Synopsys and several different ways to synthesize a given design. By experimentation, the combination of options was found that resulted in the best clock speed for this design.

The first major option is the use of the Design Compiler versus the use of the FPGA Compiler. The Design Compiler is a general purpose hardware compiler

whereas the the FPGA Compiler is targeted at lut-based FPGAs. Theoretically, the FPGA Compiler should work better for Altera parts since they are lut-based. In reality, that was not the case for this design. This was attributed to the fact that both the FPGA Compiler and the Design Compiler depend on libraries that are supplied by Altera. If those libraries are not well constructed, Synopsys may not produce the best results possible. The FLEX 10K parts were fairly new at the time this project was designed and the libraries seemed to be lacking some refinement.

Another difference in the way a design can be compiled is global compilation versus partitioned compilation. This means that the entire circuit can be compiled and optimized as a whole, or the different modules of the design can be compiled and optimized separately and then linked together. For this project, the modular compilation produced much better results. Compiling by modules yielded a clock frequency increase of more than 30% and a small reduction in area over compiling globally.

7.2.2 MAX+PLUS II

It was hoped that the MAX+PLUS II stage could be basically ‘‘push button’’, that is, little user intervention required with the tool. To get good results, however, manual placement was necessary. The placing algorithm for MAX+PLUS II seems to scatter related design elements across the FPGA. This may be to prevent congestion in any one part of the chip. For this design, however, this algorithm did not work well. Better results were obtained by directing the tool to place related components near each other. In fact, the final design would not completely route without this intervention.

Chapter 8

SYSTEM DESIGN AND SCALING ISSUES

In Chapter 5, a configuration was described with eight chunk processors on a 10K50. The results in Chapter 7 were based on this configuration. This system is discussed more extensively in this chapter, and specifically, how this system might change as FPGAs increase in their logic capacity. Another issue discussed in this chapter is how several FPGAs should be connected together along with memory for a multi-chip system.

8.1 Single FPGA Configuration

The number of processors that fit on one FPGA is limited by:

- the logic and routing available on the FPGA,
- the amount of on-chip RAM available on the FPGA, and
- the amount of off-chip RAM that can be accessed by the FPGA.

The eight processor system on the 10K50 is limited by internal routing. There are two unused EABs and enough pins to access the off-chip RAM. There are also more logic elements available, but the routing is too congested to allow more chunk processors.

The eight-chunk processors system was described in Chapter 5 and shown in Figure 5.3 which is repeated here as Figure 8.1. It shows a block diagram of the eight processor system that fits on the 10K50.

One part of the design not shown in the figure is the reconfiguration circuitry. This is the circuitry that loads new template data onto the FPGA when needed. The data that needs to be loaded is the EAB template data and the template constants. The EAB template data consists of pixel offsets for the shapesum and masks for the shapesum, brightsum, and surroundsum. The constants that have to be loaded are the $thresh_{min}$, $thresh_{max}$, $bright_{min}$, and $surround_{min}$ from Section 2.2.5.

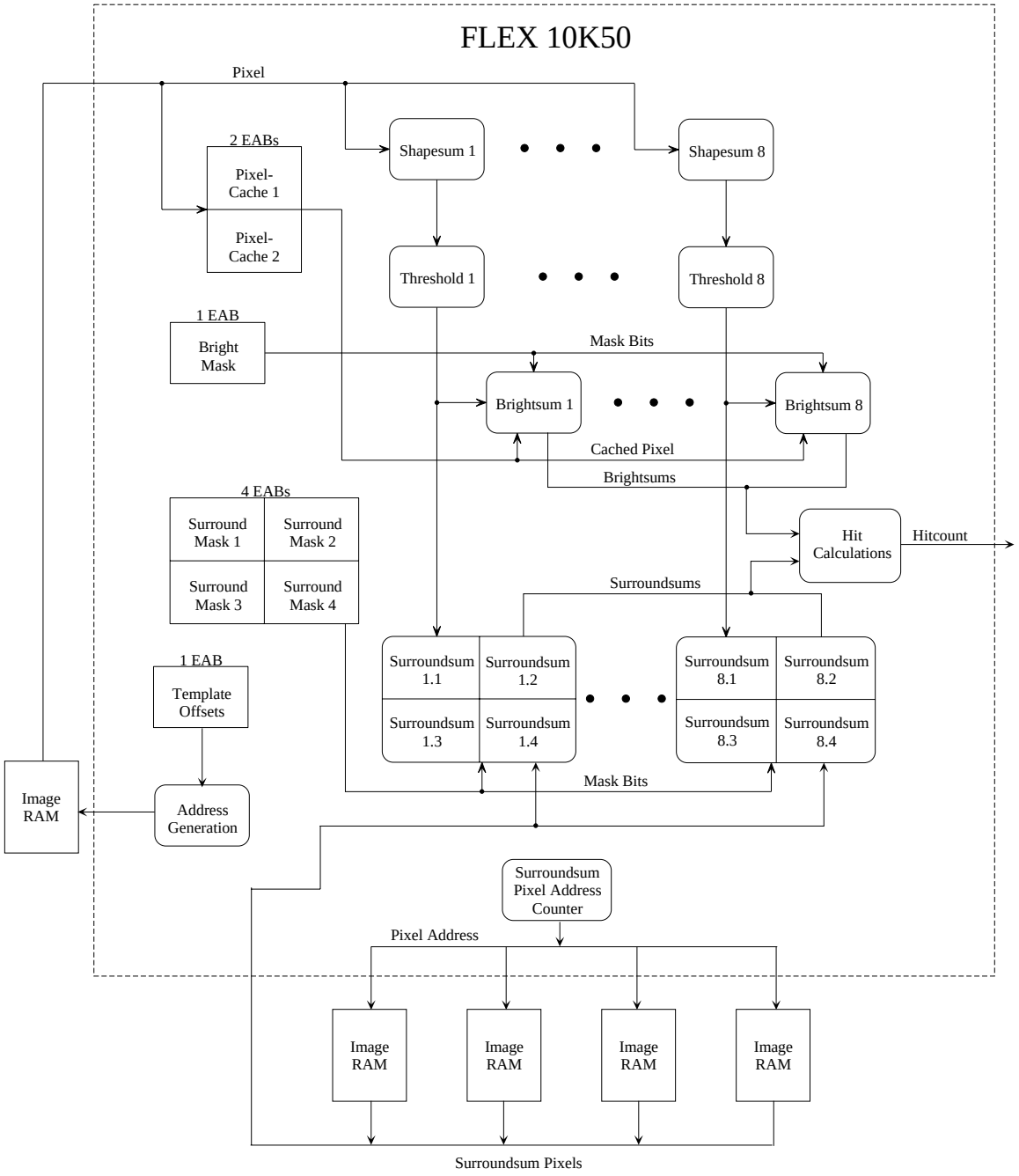


Figure 8.1: Block Diagram for Processing Group of Eight Templates

8.2 Multi-FPGA System

With the inherent parallelism in the ATR algorithm, it is simple and beneficial to scale the system to several FPGAs running in parallel. The number of FPGAs that can be included in a system is not limited by the algorithm, but by the available hardware.

Some grouping of the FPGAs in a multi-chip system is desirable. The results from all the templates in a class have to be combined to form a score for the class, so a class of templates should be grouped to run together. For example, with eight chunk processors on a chip, a group of five chips can be formed to process one class all at once. Figure 8.2 is a block diagram of such a system.

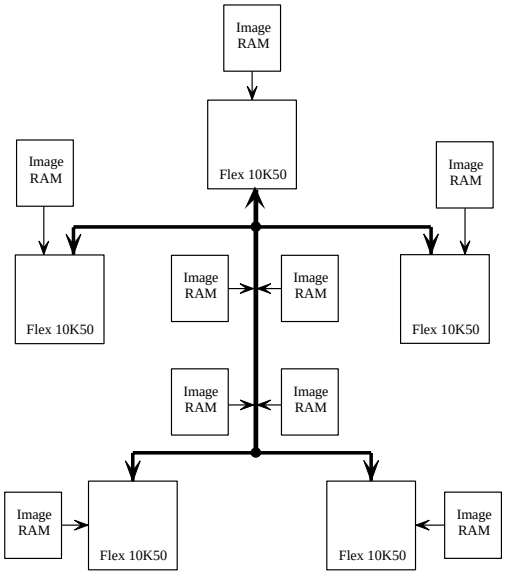


Figure 8.2: System for Processing One Class (40 Bright, 40 Surround Templates)

Running several FPGAs together has the advantage that off-chip RAM can be shared between FPGAs so the RAM to FPGA ratio goes down. As the diagram illustrates, the same four image-RAMs can be used by all five 10K50s for their surroundsum operations. A single FPGA system requires five off-chip RAMs but a five FPGA system requires only nine off-chip RAMs. The number of RAMs required is $n + 4$ where n is the number of FPGAs in the system. This is because each FPGA requires one RAM for the shapesum, but all FPGAs in a group can share the four RAMs

for the surroundsum.

8.3 Scaling Up the Design

As the density and size of FPGAs increase, the question arises of how best to configure the system to take advantage of the increased computational capacity, that is, what is the best way to scale up the design for a larger part? There are numerous small issues and questions involved here, but fundamentally, there are two choices: continue to add processors to the group, or fix the group size and place multiple groups on the FPGA.

8.3.1 Brute Force Method

To decide the best method, it is helpful to look at extreme cases. First, assume the group size is increased to some very large number. As more and more processors are added to the group, the number of on-bits in the master template increases also. Theoretically, eventually all 256 locations in the master template could be on. This means that 256 memory accesses would be required for every shapesum. The master template would not be necessary since every pixel under the template would be fetched. At the same time, each template would still have only a few of its bits on, meaning that the circuitry for the shapesum and brightsum would lie idle most of the time. One of the goals of this project was to overcome this inefficiency.

Although it seems inefficient to have idle circuitry, further work could be done to determine if a system could be developed that reads all pixels under the template and has better performance than the microcode system. Such a system could read all the pixels under the template and use the EABs as masks for the shapesum, brightsum, and surroundsum for each template. The benefits of such a system would be no performance dependency on the template data and fewer off-chip RAMs. The master template would be done away with so it would not matter if the templates combined well or not. Only one off-chip RAM would be needed to store the image for the shapesum. The pixels could be cached in an EAB and used for the brightsum and surroundsum. Or, several off-chip RAMs could be used with several pixels being read each cycle. This means that several regions of the template would be calculated in parallel, which is the way the

surroundsum is implemented in this project.

Further work would have to be done to determine if this ‘‘brute force’’ method would really be better than the microcode method. By comparing it to the microcode method, though, it seems that it would not be better, for the following reasons. As already noted, the surroundsum for the microcode version is already ‘‘brute force’’ with four regions computed in parallel. Since the two methods are equivalent in that respect, we need compare only the shapesum, threshold, and brightsum of each method. The brute force shapesum, threshold, and surroundsum require essentially the same circuitry as the microcode version, but without the address generation circuitry that reads template bit offsets and calculates pixel addresses. This reduces the area somewhat for the brute force method at the expense of clock cycles. All 256 pixels under the template would have to be fetched. To achieve the same performance as the microcode method, as measured in clock cycles, the brute force method must be made to process two or possibly four pixels at a time. To accomplish this, the circuitry would have to be duplicated to make two or four shapesum, threshold, and brightsum calculators. The trade-off, then, is address generation circuitry, which includes one EAB, versus more shapesum, threshold, and brightsum calculators. The address generation circuitry is relatively small so it is probable that the microcode version would be smaller.

Of course, the brute force implementation could be made smaller by not calculating four template regions in parallel. This would make it smaller than the microcode version and would require two to four times as many cycles to process a chunk. The advantage that the brute force method has, again, is that it does not depend on template characteristics for its performance, and it requires fewer off-chip, and possibly on-chip, RAMs.

8.3.2 Group Size

Assuming that the brute force method is not used, a group size needs to be determined to implement the microcode method on larger devices. The 10K50 is limited to eight processors so eight seemed to be a good number for the group size. For larger devices, however, the decision is not so clear. The group size could be kept at eight, or it could be increased, or it could even be decreased. There are many factors that influence the decision.

First, consider the hardware issues. Currently, the system uses eight chunk processors in a group with one group on a 10K50. Now suppose a device becomes available that fits 16 chunk processors on a chip. Should the 16 processors be combined into one group, with shared memory, or should two groups of eight be formed? First, look at the on-chip RAM requirements. For two groups of eight, 16 EABs are required. For one group of 16, only 13 EABs are required. The logic requirements for two groups will also be slightly higher than for one group. The off-chip RAM for one group is five image RAMs, the same as for a group of eight. The number of image RAMs required for two groups of eight is six, since the surround image RAMs can be shared among groups as in the multi-chip system. Table 8.1 summarizes these numbers.

Table 8.1: Hardware Requirements for 16 Chunks

Group Size	Groups	EABs	Off-Chip RAMs	LEs
8	2	16	6	≈ 4600
16	1	13	5	≈ 4400

Second, consider the data issues. Two groups of eight will require approximately the same number of cycles as one group of eight. Using one group of sixteen, however, will generally increase the cycle count. How much of an increase it will make is very dependent on the actual templates. The only way to estimate presently is to look at the synthesized templates in Appendix A. An example of how a performance figure can be calculated is presented later.

What all this says is that two groups of eight processors achieve better performance than one group of 16, at the expense of more hardware. To say which method is better is difficult. One possible metric that could be used to help decide is the Area $\times$ Time metric or A-T. [2]. A-T should not be considered the authoritative answer, but it can be used as a guide in deciding which method to use.

There are several difficulties with using A-T to calculate the performance in this situation. One is determining the time variable, since the number of clock cycles (time) required for computation is dependent on the template data. It is necessary to resort to the synthesized templates to get some idea of how many clock cycles are

required. Another difficulty is determining the area, since there are several area factors involved. Should the number of EABs be used as the area, or the number of off-chip RAMs, or the number of logic elements, or some weighted sum of all three?

To begin somewhere, take each of the possible area measurements and make an A-T calculation for each. For the time factor, use the clock cycles for the parameters $\alpha = -4.8$, $\beta_{1,*} = 2.4$, and $\beta_{2,*} = 1.0$. For 20 on-bits per template the 8-group has approximately 50 on-bits in the master template while the 16-group has approximately 70 on-bits. Table 8.2 shows the resulting A-T results using these figures. As can be seen,

Table 8.2: A-T Comparison

Area Measurement	Templates per Group	Area	Time	A $\times$ T	difference
EABs	8	16	70 cycles	1120	
	16	13	85 Cycles	1105	-1.3 %
Image RAM	8	6	70 cycles	420	
	16	5	85 Cycles	425	1.2 %
LEs	8	4600	70 cycles	322,000	
	16	4400	85 Cycles	374,000	16 %

the difference in performance measured in this way is slight. Given the approximations used in deriving the area and time factors, the difference is insignificant. For different MRF parameters and a different number of on-bits, however, the differences might be greater.

What can be concluded from this example is that the optimum group size is dependent on template data. Smaller groups will have better performance at a higher cost of hardware. Just how much better the performance depends on the template data. Until more information about the templates is available, it is difficult to specify a best group size exactly, but a good starting point is to make the group size just small enough that the number of on-bits in the master template is around 50. This will result in the threshold, brightsum, and surroundsum using about the same number of cycles so the hardware is utilized most efficiently. If the group size is increased sufficiently, the performance will approach that of the brute force method as more and more of the

master template bits are turned on.

Practically speaking, the group size should be a divisor of 40 so that templates are calculated along class boundaries. This is beneficial because the hitcount calculation uses the results from all the templates in the class. If the group size divides 40, then a class can be calculated either in parallel using multiple groups or sequentially by reconfiguring between groups. The hitcount for the class can then be calculated.

8.3.3 Scaling Projection

What will a system using this implementation look like several years from now? Obviously, FPGAs will have greater resources in the future. For a projection, assume that the FPGAs available are five times the size, in resources, of current FPGAs. For the FLEX 10K series, this means that the equivalent of the 10K50 will have 14,400 LEs and 50 EABs with the accompanying routing, assuming the ratio between LEs, EABs, and routing is maintained.

What does this mean for this implementation? With a device five times the current size of the 10K50, five groups of eight chunk processors would fit on one FPGA. That is enough to process an entire class at one orientation on one device at once. The requirements on the FPGA, shown in Table 8.3, would be 10,000 LEs, 40 EABs (assuming a 256×8 EAB), nine image-RAMs, approximately 200 user I/O pins, and sufficient routing. If a group size of ten were chosen instead of eight, less hardware would be required. The EAB requirement would be reduced to 34, the off-chip RAM reduced to eight, the I/O pins to approximately 180, and the required LEs would be somewhat fewer, though not a significant number. This hardware reduction would come with a possible increase in cycle count if the master templates of the groups had more on-bits than the eight-template groups.

Table 8.3: Resource Requirements for a 5$\times$ Device

Group Size	LEs	EABs	Image RAMs	I/O Pins
8	10,000	40	9	200
10	$< 10,000$	34	8	180

The conclusion to draw from this is that to predict performance for larger devices, a good estimate is to scale the present design results by the difference in size of the larger FPGA. A device five times bigger than the current device will have roughly five times better performance. The assumptions that are made with this estimate are that the current relative numbers of EABs, LEs, and routing are kept the same. This is a reasonable assumption. In contrast, it is not reasonable to assume that user I/O will keep up since I/O pins usually do not scale with the rest of the device. This should not be a problem for the foreseeable future since the pin count for this implementation is relatively small. Another assumption made is that the current implementation with groups of 8-10 templates is maintained. Currently, this appears to be the best configuration.

Chapter 9

SUMMARY AND CONCLUSIONS

9.1 Summary

This thesis presents results of a project to research the use of the Altera FLEX 10K with Automatic Target Recognition as the application. The stage of ATR that was implemented is chunky SLD. Chunky SLD involves template matching with 8-bit gray-scale images and binary templates. The bright templates used are sparsely populated which opens up possibilities for optimizing the implementation. The major operations in chunky SLD are the shapesum, threshold, brightsum, and surroundsum.

The Altera FLEX 10K is an SRAM based family of FPGAs. The 10K is particularly interesting because of the large embedded RAMs on chip. These RAMs open up new possibilities for applications and performance in the area of configurable computing. One goal of the project was to use these RAMs, and in particular, to use them in the control section of the circuit.

The most noteworthy part of the chunky SLD implementation developed at BYU is the shapesum and brightsum. These operations are similar to 2-D cross correlation but due to the sparseness of the bright templates they can be performed in significantly fewer than n^2 operations. The method developed for this project to implement the shapesum and brightsum stores relative offsets to template on-bits in an EAB. These offsets are added to a base offset which allows direct access to the image pixels that are needed for the operations. The surround template is divided into four quadrants and all quadrants are calculated in parallel.

Chunky SLD is a highly parallel algorithm. To extend the implementation so that multiple templates could be processed in parallel, bright templates are combined into a master template. This makes it possible to share the EABs and off-chip RAM between several chunk processors.

Because the performance of the implementation is dependent on the actual template data, and because sufficient template data was not available, it was necessary

to synthesize templates for evaluation purposes. This provided a guideline for predicting performance of this implementation.

A system composed of five FLEX 10K50s and nine image RAMs was described. The performance of the system was measured in the time required to process one Q. Using the minimum 70 cycles per image offset, this system is predicted to process one Q 852 ms.

As larger devices become available, the question arises of how to scale the design to take advantage of the greater capacity. It was determined that the number of chunk processors in a group should be kept small---probably 8-10---for best performance. As more FPGA area becomes available, more groups can be added to the FPGA.

9.2 Conclusions

Section 1.3 listed three major goals of this project. They were to

- Exploit the sparseness of the bright templates,
- Utilize the on-chip RAM of the Altera FLEX 10K FPGA, and
- Demonstrate the use of embedded RAM for control.

These were met in the following ways.

Exploit Sparseness

The bright templates are sparsely populated. Since the algorithm requires that only the image pixels under the on-bits in the bright template be used for the shapesum and brightsum, it was determined that better performance could be achieved by loading only those pixels from the image memory. This was accomplished by designing a system that stores relative offsets to the on-bits in the on-chip RAM. This makes it possible to calculate the address of a necessary pixel every clock cycle. This significantly reduces the number of memory accesses required to perform the shapesum and brightsum. The surroundsum was not implemented with this method because the surround templates are not sparsely populated. Instead, the surround template was divided into four quadrants with all quadrants calculated in parallel.

Utilize On-Chip RAM

Altera is one of the first FPGA makers to dedicate resources strictly to RAM. Other FPGAs allow configuration of logic cells as RAM but do not have dedicated RAM. Because the RAM is specifically designed as RAM and not as general logic hardware, it is more dense which means it has a much higher capacity than the logic cell RAM of other devices. This project uses between four and five kilobytes of the available 20 kilobytes of embedded RAM on the 10K50. If logic cells were used to implement this RAM instead, 250 to 300 four input LUTs would be required. This is a significant fraction of the available LUTs on current FPGAs. The routing requirement for the LUT RAM addressing would also be significant. The 10K50 EABs were used for the following purposes:

- The offsets to the on-bits of the bright templates are stored there,
- Mask bits are stored which control the operation of the accumulators and counters in the shapesum, brightsum, and surroundsum, and
- Image pixels are stored there as they are loaded for the shapesum so that they do not have to be reloaded for the brightsum.

Embedded RAM for Control

FPGA RAM can be used for many things, but most uses and proposed uses are for some type of data buffer. The attempt in this project was to use the EABs on the 10K for control, rather than or in addition to using them for temporary data storage. This was done by storing the template information in the EABs. The offsets and mask bits that were stored in the EABs can be viewed as microinstructions for the chunk processor. With each new template to be processed, the EABs are loaded with new microinstructions. This, in effect, reconfigures the device for new templates.

The performance of this system is good when compared to other implementations of the same algorithm on different platforms. Table 7.3 compares the implementation on the Altera 10K50 with the implementations on two other platforms and shows that the Altera performance is as good or better than other systems, using the figures currently available.

Perhaps the biggest drawback of this implementation is its dependency on template data for performance. If the template data is favorable, this is actually not a drawback but a strength. The template synthesis gives some indication that templates will not likely combine as well as hoped. This lowers performance somewhat, but not enough to make the implementation unusable. If the templates combined so poorly that the number of clock cycles doubled, this would double the time required to calculate one Q. As Table 7.3 shows, the performance would still be good relative to other platforms.

This project was successful in many respects. The project goals given in the introduction were attained. This was done while at the same time achieving good performance. Although finding the best implementation for chunky SLD was not one of the primary goals of this project, the implementation presented here is definitely a candidate for use in a real system. Beyond that, it demonstrates an innovative use of embedded RAM in FPGAs and shows that large, embedded RAM is a valuable FPGA commodity for some applications. Finally, it is yet another example of how FPGAs can be used with good results for real-world, computationally intensive problems.

Appendix A

TEMPLATE SYNTHESIS RESULTS

This appendix presents the details of the template synthesis discussed in Chapter 6. First, Markov Random Fields [8] are discussed, then they are related to templates. After that, the algorithm is presented that was used to generate the template data. Finally, the results of the synthesis are given.

A.1 Template Model

The bright templates were modeled with Markov Random Fields.

A.1.1 Markov Random Fields

A Markov random field is a statistical model for two dimensional data. It is a joint probability density over a two dimensional field where the probability of any location, $X(m,n)$, in that field taking on a given value is dependent only on the values of the neighbors of $X(m,n)$. In other words, it is a limited joint probability density. The limitation is that the joint probability is only between a location and its neighbors, and not between a location and all locations in the entire field.

The *order* of the MRF determines the number of neighbors used in the joint probability calculation. A first order MRF uses closest neighbor dependency while a higher order MRFs extend the dependency to more distant neighbors. Figure A.1 shows the neighbors of a position X that are included in the probability calculation for that position. For example, a first order MRF would include the locations marked with a ‘‘1’’ in calculating the probability that the location has a certain value. A second order MRF would include the locations marked ‘‘1’’ and ‘‘2’’, and so on.

A.1.2 MRFs and Templates

For template modeling, a MRF works as follows. The template, or random field, is binary so the only values that a pixel can be are ‘‘1’’ and ‘‘0’’. What the MRF

	4	3	4	
4	2	1	2	4
3	1	X	1	3
4	2	1	2	4
	4	3	4	

Figure A.1: Nth Order Neighbors for a Markov Random Field

model says is that the probability of any pixel being a ‘‘1’’ depends on the values of its neighbors. In general, a MRF could either stipulate that if surrounding pixels are ‘‘1’’s, a pixel is more likely to be ‘‘1’’, or it could stipulate that the pixel is more likely to be ‘‘0’’. It can be even more complicated by specifying higher probabilities for some neighbors and lower probabilities for others.

For a template, however, a pixel is more likely to be ‘‘1’’ if its neighbors are ‘‘1’’s. This is because of the hypothesized cluster feature of the templates. If the parameters of the model are chosen to reflect this idea, the result is that on-bits will be more likely found in groups than spread out. The other two criteria, sparseness and centering, are taken care of by the MRF generation algorithm, and not specifically by the Markov model. A MRF does not allow providing positional information within the field.

The MRF does not specify what probability distribution is actually used to calculate the probability of a certain pixel. The distribution is something that must be chosen. The one used for template generation is a binomial model which is discussed in [9].

A.2 Synthesis Algorithm

The algorithm used to synthesize the templates comes from [9]. This algorithm is designed to generate Markov random field gray-scale images. For template generation, the image has just two possible pixel values. The algorithm works by first generating a completely random image and then moving pixels around in the image

until a state is reached where the image meets the Markov criteria.

The heart of the algorithm is the switching routine. This routine continually and randomly chooses two pixels in the image and swaps them if the resulting image will be ‘‘better’’ than the image without these two pixels swapped. ‘‘Better’’ means that one image has a higher joint probability than the other. When this swapping is done enough times, the resulting image parameters meet the desired parameters to within some small error. There is a little twist that keeps the images from converging to the same image each time. The two pixels may be swapped even if the resulting image is not better than the previous image. This swap is based on a random toss of the die.

Before applying the switching routine, an initial image must be generated. This can be any image with the number of on-bits desired. It is this initial image that controls the number of on-bits to meet the sparse requirement of the templates. The switching routine swaps pixels but it does not create new on pixels or delete old ones. That means that if a template with 10 on-bits is to be synthesized, then an initial template with any 10 bits turned on should be input to the switching routine. Practically speaking, the initial image should be random. That is, a template with 10 (or however many) random bits turned on should be generated. After applying the switching routine, the image will still have 10 bits on but they will be rearranged.

The last piece that is needed for the MRF generator is a probability distribution. This is used to calculate the probability that a given bit is on. As will be seen, the order of the MRF as well as the strength of the clustering is incorporated into the probability function.

The algorithm is discussed in more detail in the following sections.

A.2.1 Switching Routine

The switching routine is shown here in C-type pseudo-code.

```
1:  while (!stable()) {
2:     randomly choose sites pix1, pix2, with pix1 != pix2;
3:     ratio = p(Y) / p(X);
4:     if (ratio >= 1.0)
5:        switch(pix1, pix2);
```

```
 6:     else {
 7:        randval = uniform random on [0,1];
 8:        if (ratio > randval)
 9:           switch(pix1, pix2);
10:     }
11:  }
```

The lines of code have the following meanings:

1. The `stable()` function returns TRUE if the number of successful switches drops below a certain percentage of total attempted switches for the current iteration. One iteration is defined as a number of attempted switches equal to the number of pixels in the image. For example, in a 16 $\times$ 16 image one iteration is 256 attempted switches. [9] shows that stability can be reached in under ten iterations. Stability can also be based on how closely the parameters of the image meet the input parameters.

2. Choose two pixel locations randomly.

3. `Y` represents the state of the image with the pixels swapped. `X` represents the state of the image without swapping the pixels. `p(Y)` is the joint probability of the image with the pixels swapped. `p(X)` is the joint probability of the image with the pixels not swapped. The method for calculating the joint probability of the image is discussed later.

4. If the new state (`Y`) is better than the old state (`X`), then `ratio` will be greater or equal to 1.0.

5. The algorithm swaps the pixels because doing so results in an image that meets the desired parameters more closely. `switch(pix1, pix2)` simply accomplishes the swap.

6. If the pixels are not swapped in the previous lines, there is still a chance they may be swapped.

7. Roll a die.

8. If the right number turns up...

9. Swap the pixels to perturb the system. This prevents the undesired effect of every image settling to the same image.

A.2.2 Probability Function

The equations for the probability functions are presented here without the theory behind them. The theoretical discussion can be found in [9].

The general equation for $p(\mathbf{Y})/p(\mathbf{X})$ used in the switching routine above is given by

$$\frac{p(\mathbf{Y})}{p(\mathbf{X})} = \prod_{i=1}^{M} \frac{p(X(i) = y(i) \mid \text{pixels in neighborhood})}{p(X(i) = x(i) \mid \text{pixels in neighborhood})}. \tag{A.1}$$

$p(\mathbf{Y})/p(\mathbf{X})$ is the `p(Y) / p(X)` in the pseudo-code above. M is the total number of pixels in the image. If the image is $N \times N$ then $M = N^2$. $X(i)$ is a random variable that represents the i^{th} pixel. $y(i)$ is the value that $X(i)$ would take on in state **Y**. $x(i)$ is the value that $X(i)$ takes on in state **X**. The method of calculating $p(X(i)...)$ is given shortly.

Equation A.1 calculates the ratio of the joint probabilities of the image with and without swapping two pixels. State **X** is the state without swapping and state **Y** is the state with swapping. Because only two pixels change between state **X** and **Y**, all terms in the product cancel except for the probabilities of the two pixels in question. The calculation for these two pixels is just the resulting probability with these two pixels switched. With this simplification, the new formula for $p(\mathbf{Y})/p(\mathbf{X})$ is

$$\frac{p(\mathbf{Y})}{p(\mathbf{X})} = \frac{p(X(i) = x(j) \mid \text{neighbors}) \times p(X(j) = x(i) \mid \text{neighbors})}{p(X(i) = x(i) \mid \text{neighbors}) \times p(X(j) = x(j) \mid \text{neighbors})} \tag{A.2}$$

Now, to calculate $p(X(i) = x(j))$ and so forth, a potential function is used. This function is

$$p(X = x|neighbors) = p(X = x|T) = \frac{e^{xT}}{1 + e^{T}}. \tag{A.3}$$

X is the random variable representing the pixel whose joint probability is being calculated. x is the value of the pixel for which the probability is being calculated. T is a parameter that is calculated from the neighbors of X.

The calculation of T is where the Markov parameters enter in. It is given by

$$T = \alpha + \sum_{i=1}^{O} T_i. \tag{A.4}$$

In this equation, α is an arbitrary constant, O is the order of the MRF, and the T_i are given by

$$\begin{aligned}
T_1 &= \beta_{1,1}(v_{1,1} + v_{1,3}) + \beta_{1,2}(v_{1,2} + v_{1,4}) \\
T_2 &= \beta_{2,1}(v_{2,1} + v_{2,3}) + \beta_{2,2}(v_{2,2} + v_{2,4}) \\
T_3 &= \beta_{3,1}(v_{3,1} + v_{3,3}) + \beta_{3,2}(v_{3,2} + v_{3,4}) \\
T_4 &= \beta_{4,1}(v_{4,1} + v_{4,2} + v_{4,5} + v_{4,6}) + \beta_{4,2}(v_{4,3} + v_{4,4} + v_{4,7} + v_{4,8}).
\end{aligned}$$

The $v_{i,j}$ are the neighborhood pixels and their position is shown in Figure A.2. The $\beta_{i,j}$ are arbitrary parameters. These, along with α, control the characteristics of the MRF. For template generation, these parameters control the strength and direction of the on-bit clustering.

	$v_{4,3}$	$v_{3,2}$	$v_{4,2}$	
$v_{4,4}$	$v_{2,2}$	$v_{1,2}$	$v_{2,1}$	$v_{4,1}$
$v_{3,3}$	$v_{1,3}$	X	$v_{1,1}$	$v_{3,1}$
$v_{4,5}$	$v_{2,3}$	$v_{1,4}$	$v_{2,4}$	$v_{4,8}$
	$v_{4,6}$	$v_{3,4}$	$v_{4,7}$	

Figure A.2: Neighborhood Pixels for Calculating the T Parameter.

It is by adjusting α, β, and the order of the MRF that templates with varying characteristics can be produced. The value of these parameters for real templates is not known.

A.3 Synthesis Code

The synthesis algorithm was implemented as a C++ program. The C++ code is included in Appendix C. The program was written to be flexible so that the MRF

parameters, the number of on-bits per template, the number of templates in a group, and the size of the templates could be altered easily. The program outputs the synthesized templates in the form of pbm files. It also calculates the the number of on-bits in the group of templates. Additional functionality was added so that the program generates MATLAB m-code which plots the number of on-bits in a group as a result of the number of on-bits per template.

A.4 Synthesis Results

Both first and second order MRFs were generated. Clustering strength in the horizontal and vertical directions was assumed to be equal. This means that $\beta_{1,1} = \beta_{1,2}$, which will be denoted $\beta_{1,*}$, and $\beta_{2,1} = \beta_{2,2}$, which will likewise be denoted $\beta_{2,*}$. First order MRFs were generated simply by setting $\beta_{2,*} = 0.0$.

The range of parameters that were tested were

- $\beta_{1,*} = 0.0...4.4$ in increments of 0.4,
- $\beta_{2,*} = 0.0...3.0$ in increments of 1.0, and
- $\alpha = 2\beta_{1,*}$.

All combinations of these values were synthesized.

The following pages show the results of the template synthesis. The results are in the form of graphs which plot the number of on-bits in a group of templates with varying parameters versus the number of on-bits per template in the group.

The parameters control the strength of the clustering of the Markov Random Field generator. α is present in the potential function for any order MRF. The β parameters are order dependent. For example, $\beta_{1,1}$ and $\beta_{1,2}$ are first order parameters while $\beta_{2,1}$ and $\beta_{2,2}$ are second order parameters. The synthesis was done with all n^{th} order parameters the same. This means that $\beta_{1,1} = \beta_{1,2}$ and so forth. The notation used on the plots is $\beta_{1,*} = c$ which means $\beta_{1,1} = \beta_{1,2} = c$. Also, α was set to be equal to $2\beta_{1,*}$.

First and second order MRF templates were synthesized. The plots do not distinguish between first and second order explicitly. This is because $\beta_{2,*} = 0.0$ is the

same as a first order MRF. To find all first order results, simply turn to the graphs with $\beta_{2,*} = 0.0$.

The horizontal axis represents the number of on-bits per template in the group. The vertical axis represents the resulting number of on-bits in a combined group of eight templates. All graphs have the same scale. There is a horizontal line on each graph at 50 on-bits in the group. This is the maximum number of on-bits for best performance with this implementation.

For each parameter set, there are two graphs. One is for eight templates in the group, and the other is for 16 templates in the group. After each two pages of graphs, there is a page of representative templates. The templates are in groups of eight. Each group of eight has the same parameters as the corresponding plot on the page previous to it. The templates have 12 on-bits each.

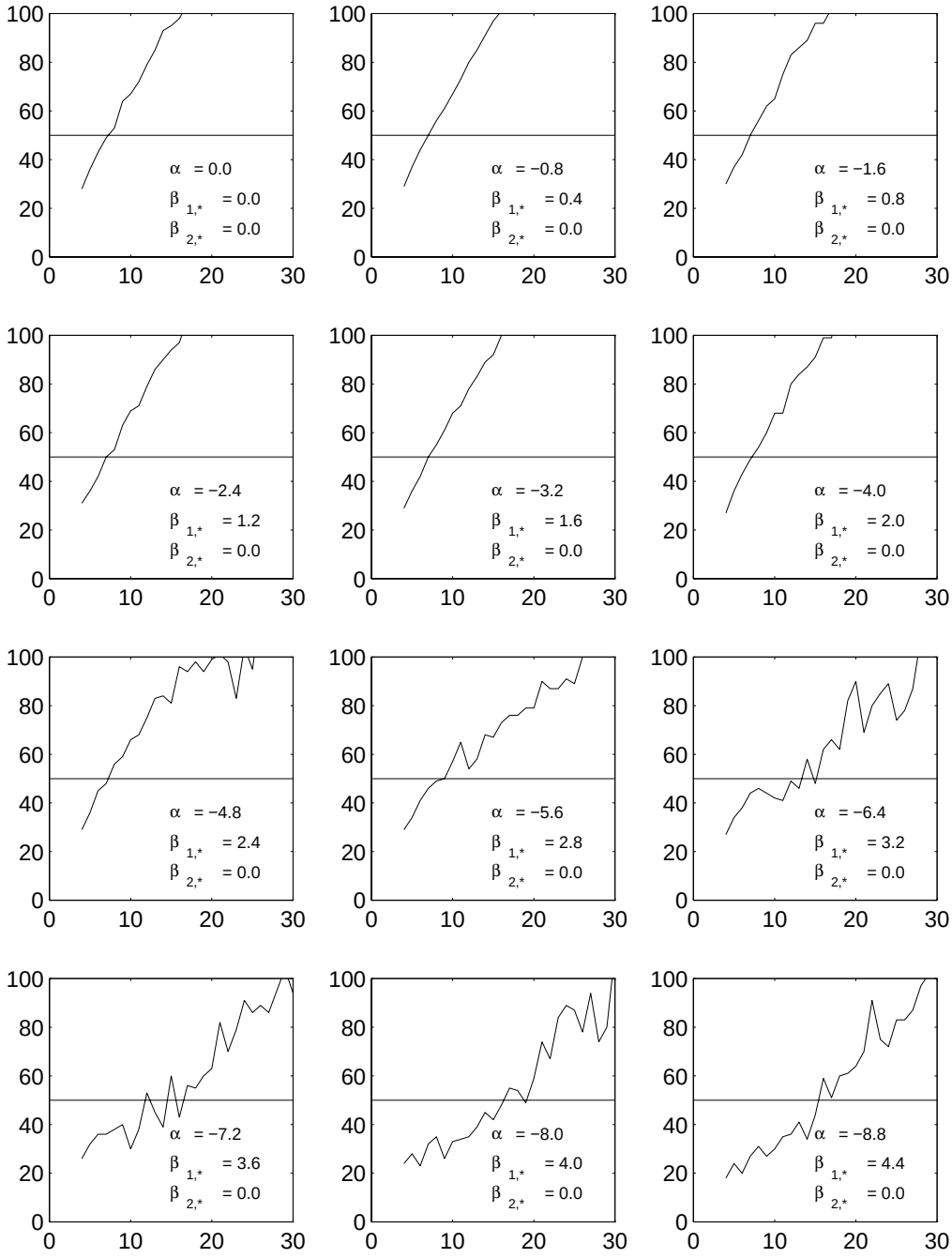


Figure A.3: First Order Graphs, 8 Templates per Group

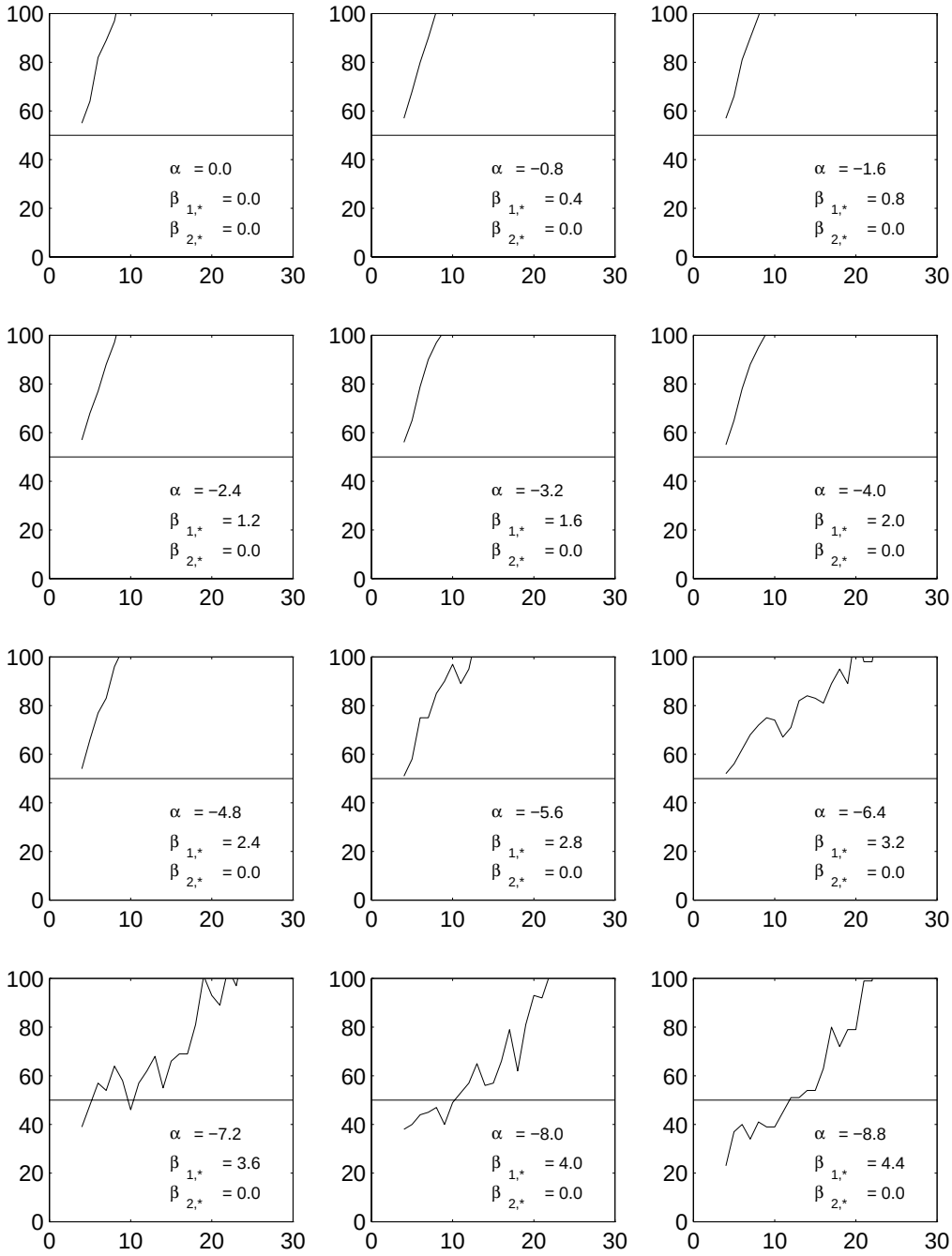


Figure A.4: First Order Graphs, 16 Templates per Group

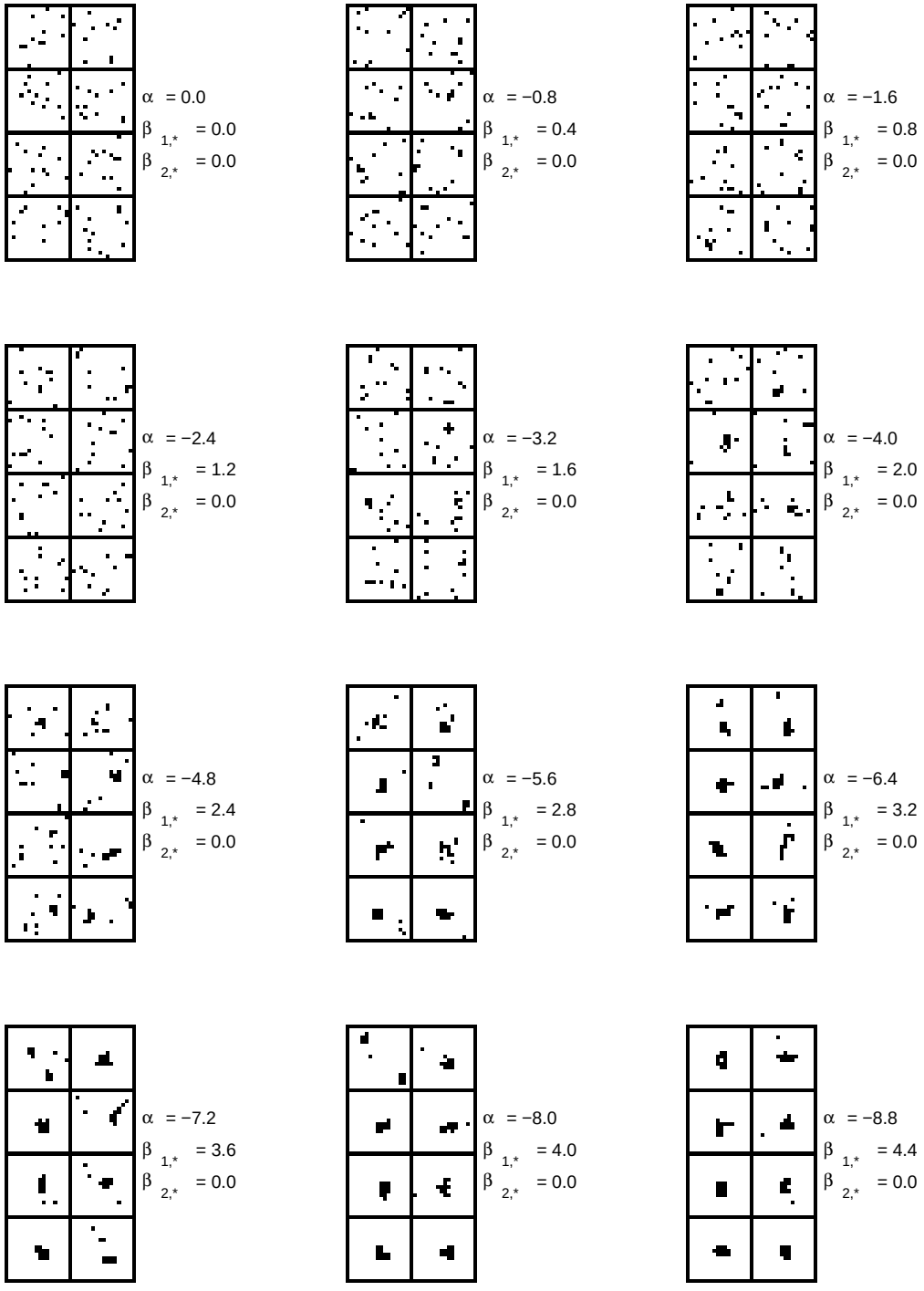


Figure A.5: First Order Templates, 12 On-Bits per Template

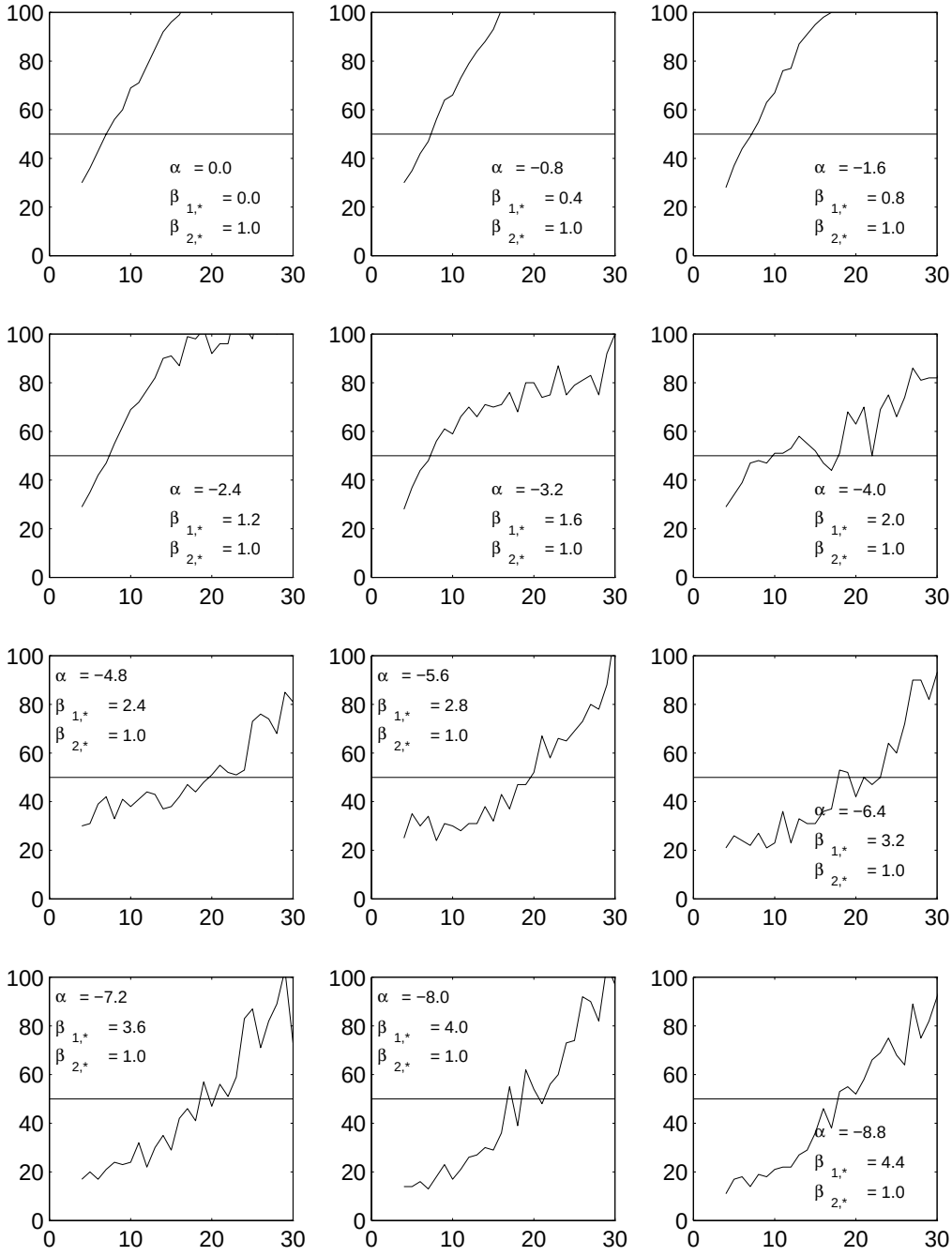


Figure A.6: Second Order Graphs $\beta_{2,*} = 1.0$, 8 Templates per Group

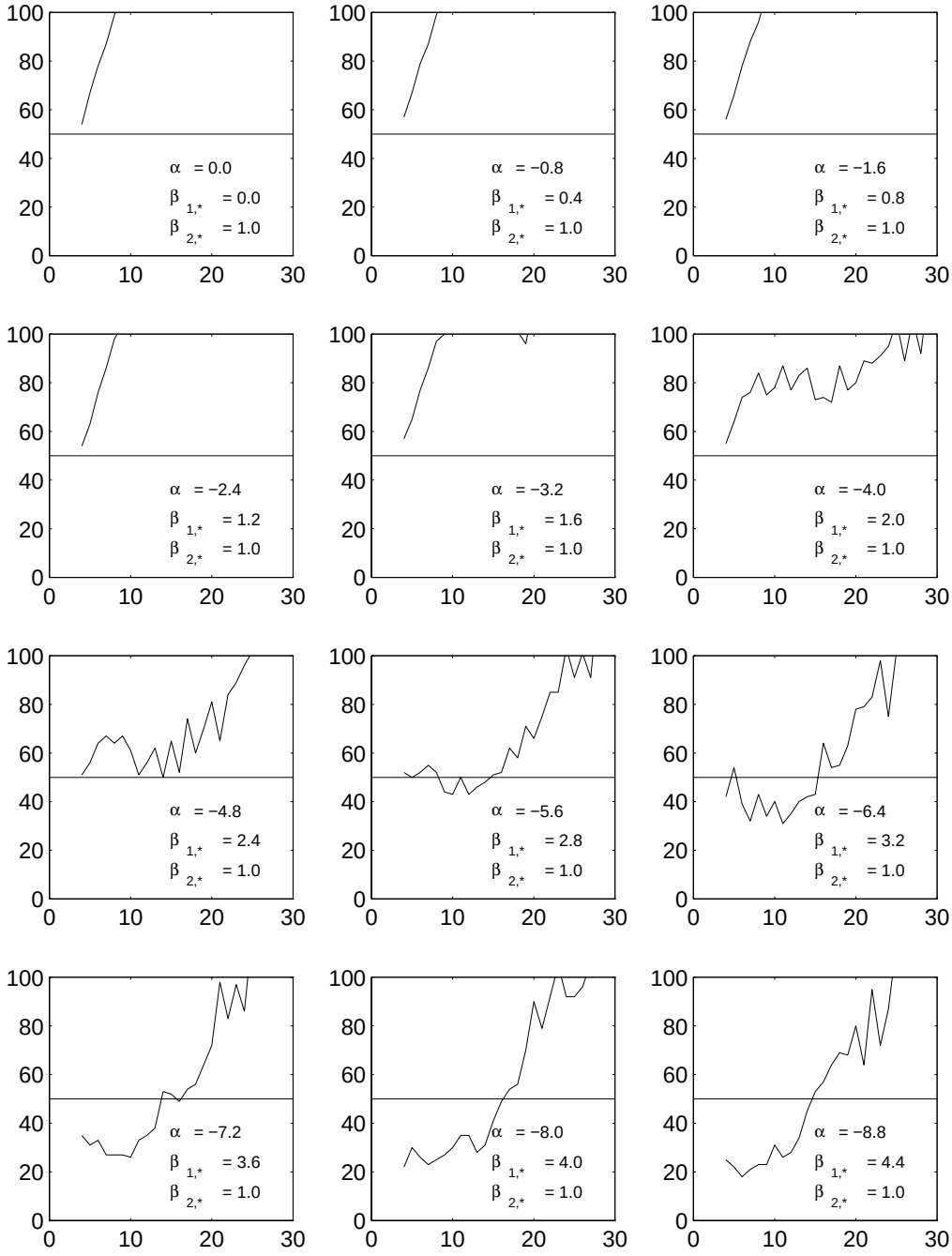


Figure A.7: Second Order Graphs $\beta_{2,*} = 1.0$, 16 Templates per Group

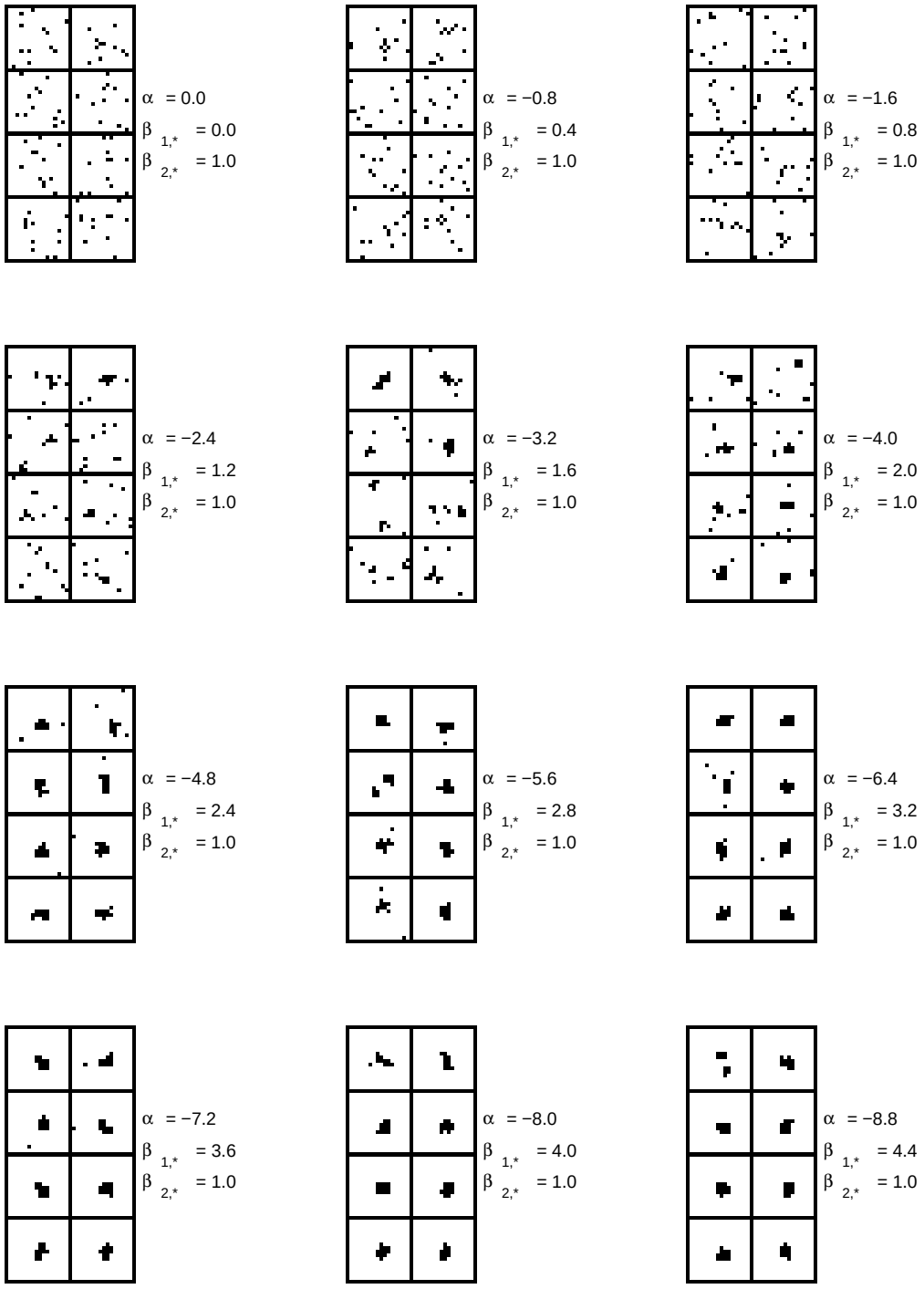


Figure A.8: Second Order Templates, $\beta_{2,*} = 1.0$, 12 On-Bits per Template

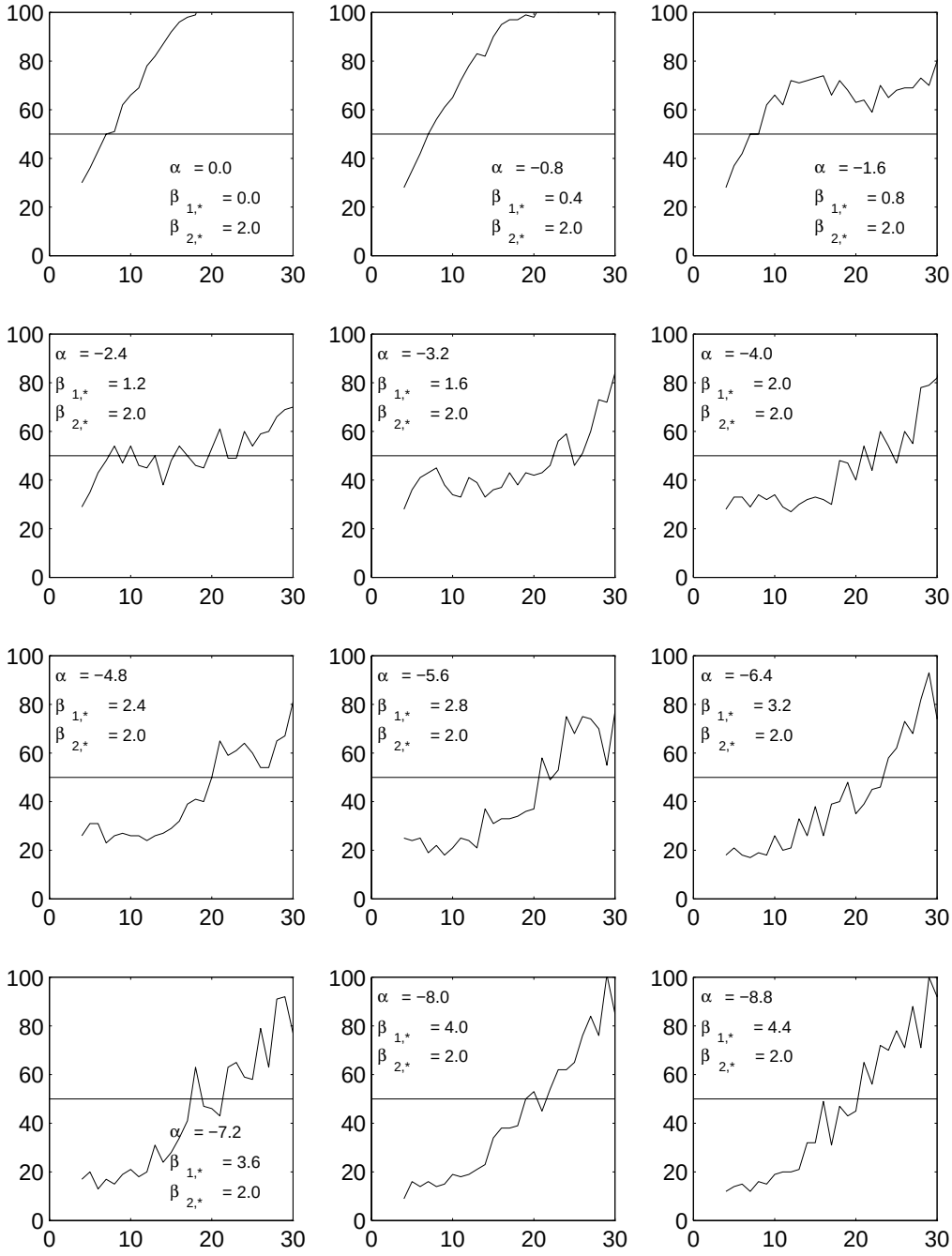


Figure A.9: Second Order Graphs $\beta_{2,*} = 2.0$, 8 Templates per Group

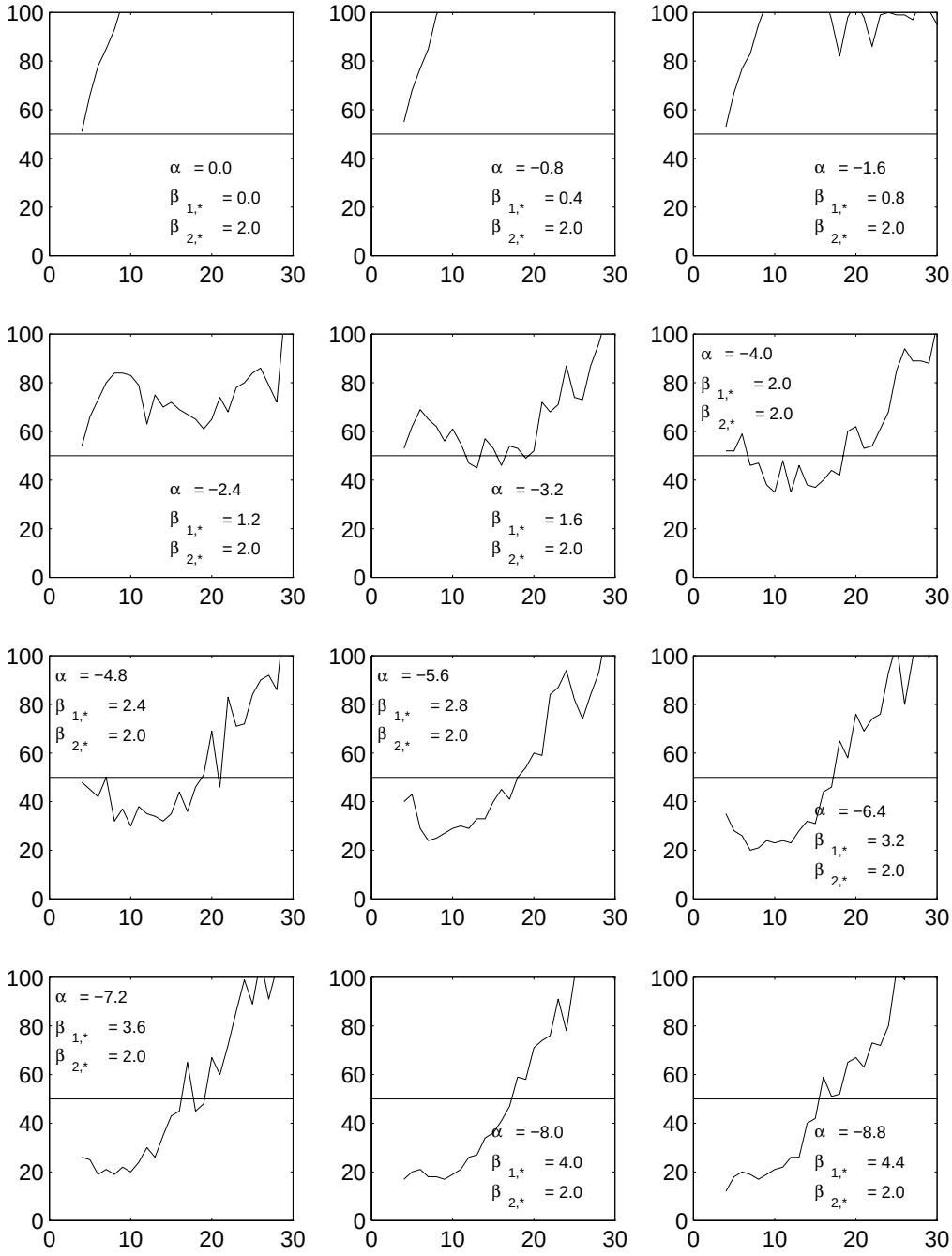


Figure A.10: Second Order Graphs $\beta_{2,*} = 2.0$, 16 Templates per Group

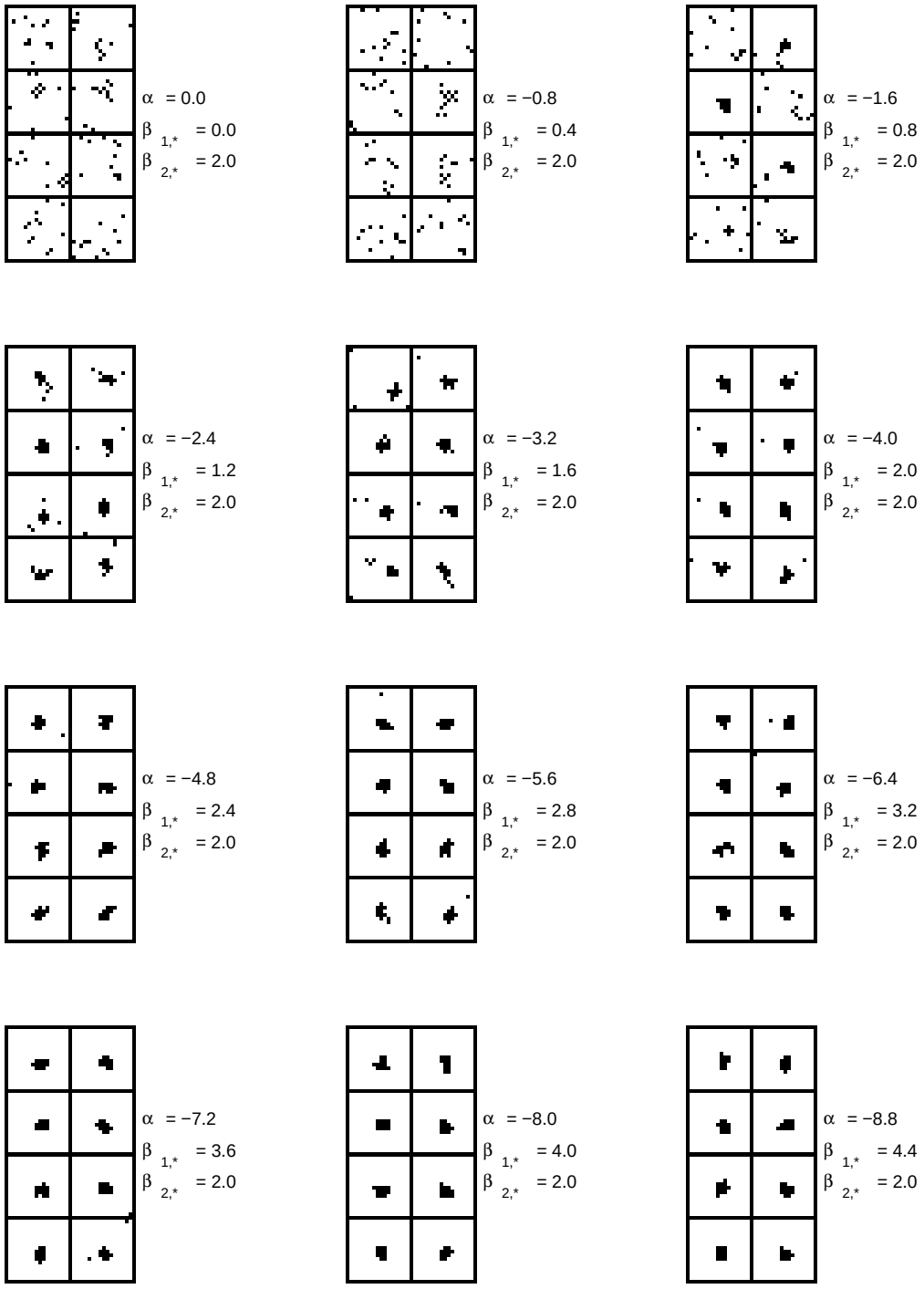


Figure A.11: Second Order Templates, $\beta_{2,*} = 2.0$, 12 On-Bits per Template

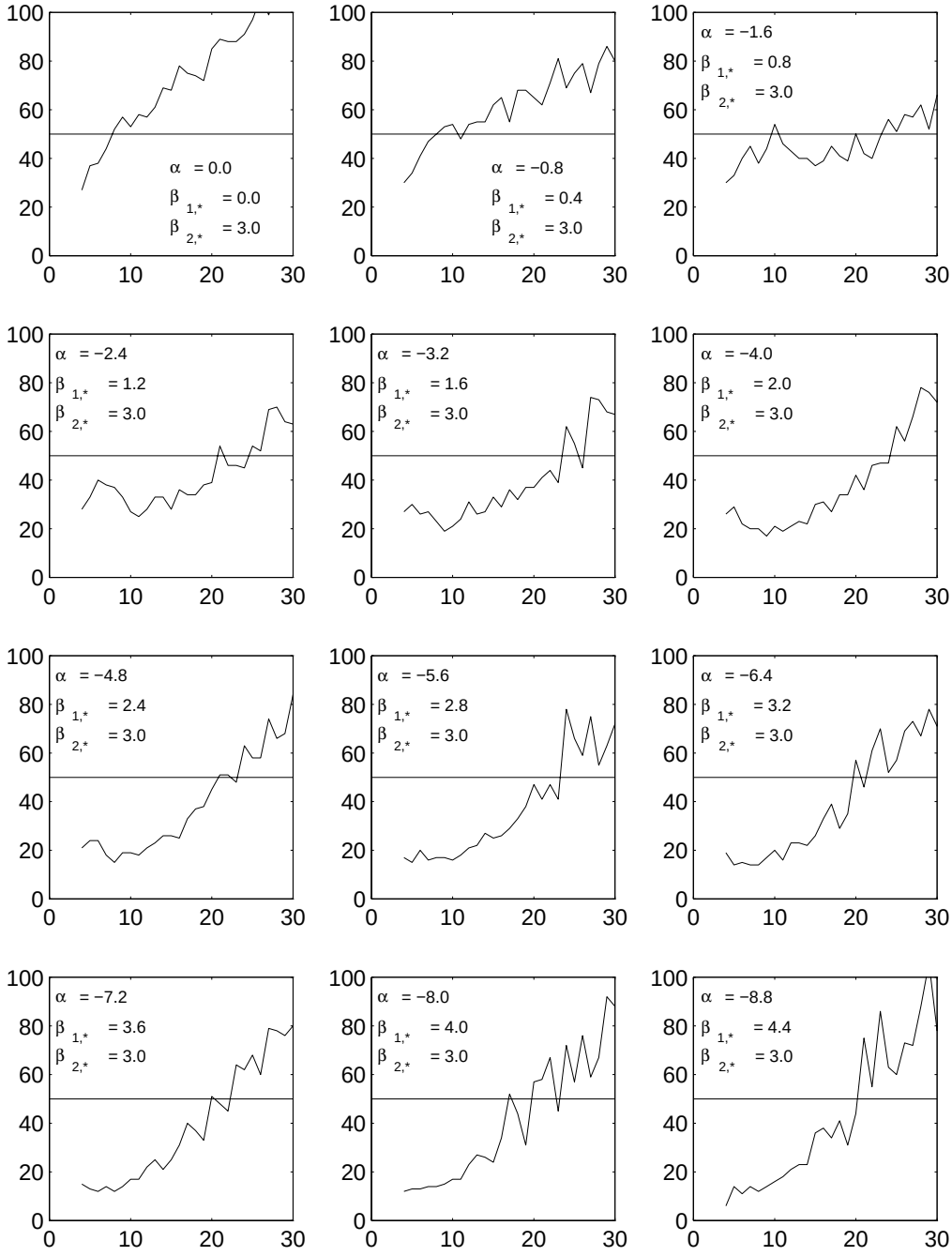


Figure A.12: Second Order Graphs $\beta_{2,*} = 3.0$, 8 Templates per Group

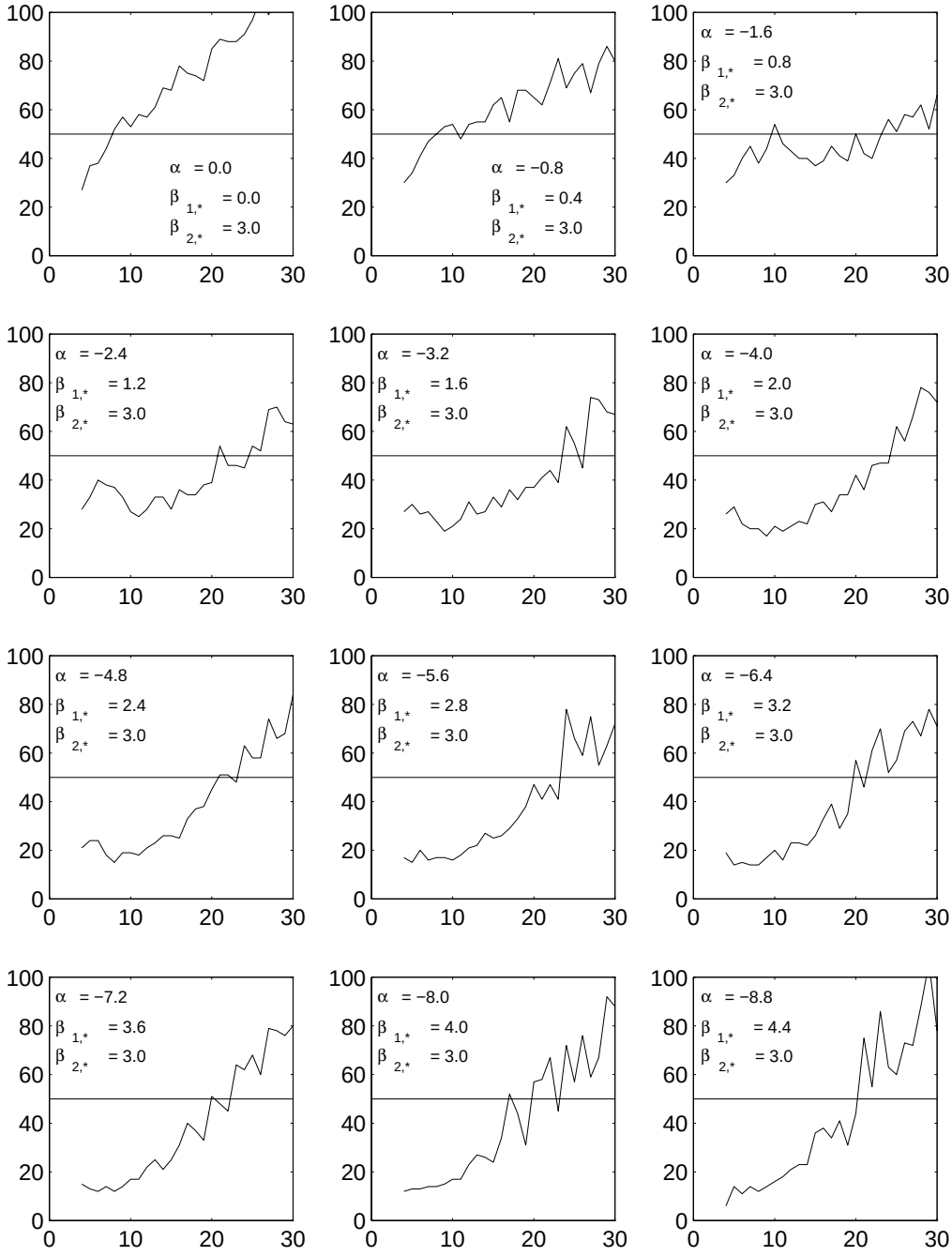


Figure A.13: Second Order Graphs $\beta_{3,*} = 3.0$, 16 Templates per Group

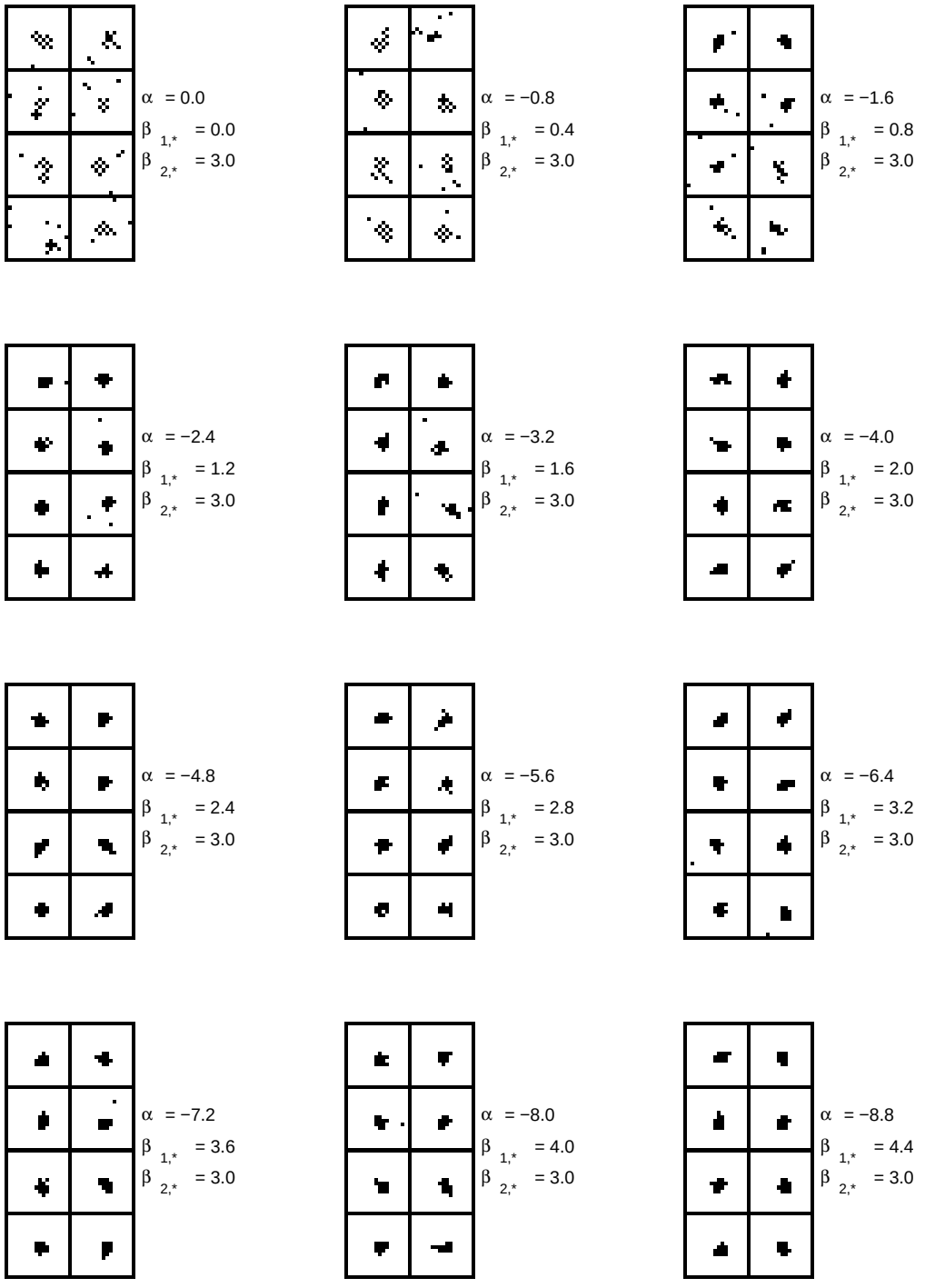


Figure A.14: Second Order Templates, $\beta_{2,*} = 3.0$, 12 On-Bits per Template

Appendix B

VHDL SOURCE CODE

The VHDL code for this project is presented here. The code has the hierarchy shown in Figure B.1.

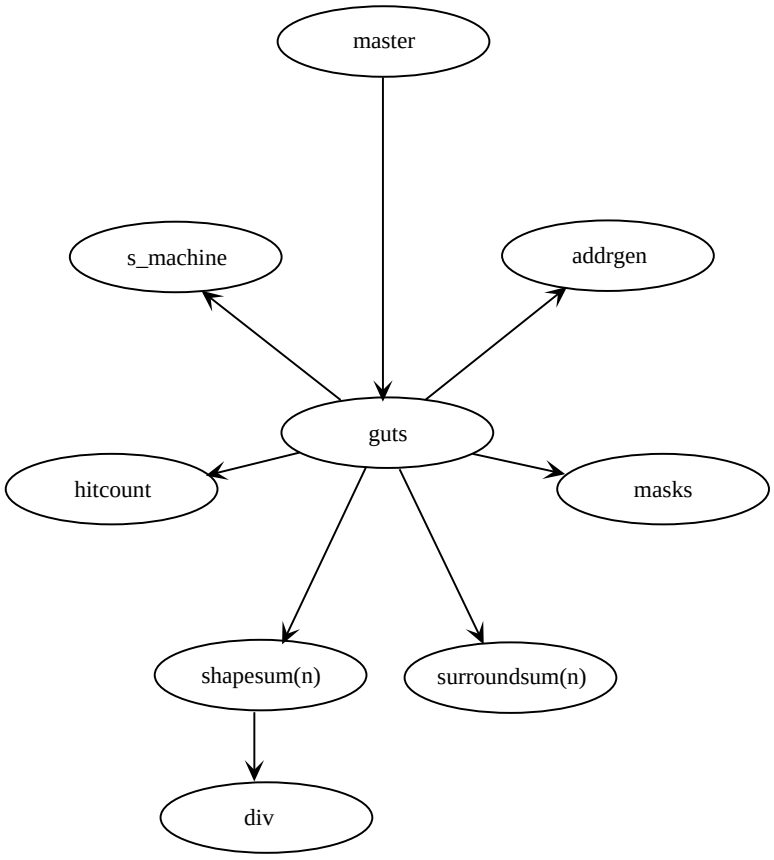


Figure B.1: VHDL Hierarchy

The modules have the following basic functions:

- `master`: Specifies the FPGA I/O. Registers most of the incoming signals.

- `guts`: Instantiates most of the sub-modules and acts as the interface between them.
- `shapesum`: Computes the shapesum, threshold, and brightsum. Instantiates `div` for the threshold divider.
- `div`: Performs the division for the threshold operation.
- `surroundsum`: Computes the surroundsum.
- `s_machine`: Implements the controlling state machine.
- `masks`: Instantiates the EABs that store the template masks.
- `addrgen`: Computes the addresses for the off-chip image RAMs.
- `hitcount`: Tallies the hits from the chunk processors.

B.1 Entity and Architecture Files

B.1.1 master

```
--------------------------------------------------------------------------------
--  Brings signals on and off chip.  Instances guts.vhd which instances
--  everything else.
--  Richard Ross
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.surr4_p.all;
use work.addr_p.all;
use work.comps.all;

--------------------------------------------------------------------------------
--  Entity Declaration
--------------------------------------------------------------------------------
entity master is
  port(clk       : in std_logic;
       globalrst : in std_logic;     --  Resets everything, including counts
       clear     : in std_logic;             --  Clears the calculations
       shpixel   : in pixel_t;
       s0pixel   : in pixel_t;
       s1pixel   : in pixel_t;
       s2pixel   : in pixel_t;
       s3pixel   : in pixel_t;
       reconfig  : in std_logic;
       func      : in func_t;
       loadaddr  : in loadaddr_t;
       loaddata  : in loaddata_t;
       clearaddr : in std_logic;
```

```
       shaddr    : out extmemaddr_t;
       s0addr    : out extmemaddr_t;
       s1addr    : out extmemaddr_t;
       s2addr    : out extmemaddr_t;
       s3addr    : out extmemaddr_t;
       newhit    : out std_logic;
       hits      : out std_logic_vector(3 downto 0)
       );
end master;


--------------------------------------------------------------------------------
--  architecture declaration
--------------------------------------------------------------------------------
architecture basic of master is

  --  registered versions of the incoming pixels
  signal shpixel_r : pixel_t;
  signal s0pixel_r : pixel_t;
  signal s1pixel_r : pixel_t;
  signal s2pixel_r : pixel_t;
  signal s3pixel_r : pixel_t;

  --  registered reconfigure signals
  signal reconfig_r  : std_logic;
  signal func_r      : func_t;
  signal loadaddr_r  : loadaddr_t;
  signal loaddata_r  : loaddata_t;
  signal clearaddr_r : std_logic;


begin


  --  EAB address counter and other registered statements
  process (clk)
  begin
    if clk'event and clk = '1' then
      shpixel_r   <= shpixel;
      s0pixel_r   <= s0pixel;
      s1pixel_r   <= s1pixel;
      s2pixel_r   <= s2pixel;
      s3pixel_r   <= s3pixel;
      reconfig_r  <= reconfig;
      func_r      <= func;
      loadaddr_r  <= loadaddr;
      loaddata_r  <= loaddata;
      clearaddr_r <= clearaddr;
    end if;
  end process;


  --------------------------------------------------------------------------------
  --  Component Instantiations
  --------------------------------------------------------------------------------
  interface : guts
    port map(clk => clk,
             globalrst  => globalrst,
             clear      => clear,
             shpixel_r  => shpixel_r,
             s0pixel_r  => s0pixel_r,
             s1pixel_r  => s1pixel_r,
             s2pixel_r  => s2pixel_r,
             s3pixel_r  => s3pixel_r,
             reconfig_r => reconfig_r,
```

```
            func_r     => func_r,
            loadaddr_r => loadaddr_r,
            loaddata_r => loaddata_r,
            clearaddr_r => clearaddr_r,

            shaddr    => shaddr,
            s0addr    => s0addr,
            s1addr    => s1addr,
            s2addr    => s2addr,
            s3addr    => s3addr,
            newhit    => newhit,
            hits      => hits
            );

end basic;
```

B.1.2 master

```
--------------------------------------------------------------------------------
--  Brings signals on and off chip.  Instances guts.vhd which instances
--  everything else.
--  Richard Ross
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.surr4_p.all;
use work.addr_p.all;
use work.comps.all;

--------------------------------------------------------------------------------
--  Entity Declaration
--------------------------------------------------------------------------------
entity master is
  port(clk       : in std_logic;
       globalrst : in std_logic;     --  Resets everything, including counts
       clear     : in std_logic;             --  Clears the calculations
       shpixel   : in pixel_t;
       s0pixel   : in pixel_t;
       s1pixel   : in pixel_t;
       s2pixel   : in pixel_t;
       s3pixel   : in pixel_t;
       reconfig  : in std_logic;
       func      : in func_t;
       loadaddr  : in loadaddr_t;
       loaddata  : in loaddata_t;
       clearaddr : in std_logic;

       shaddr    : out extmemaddr_t;
       s0addr    : out extmemaddr_t;
       s1addr    : out extmemaddr_t;
       s2addr    : out extmemaddr_t;
       s3addr    : out extmemaddr_t;
       newhit    : out std_logic;
       hits      : out std_logic_vector(3 downto 0)
       );
end master;


--------------------------------------------------------------------------------
--  architecture declaration
--------------------------------------------------------------------------------
architecture basic of master is
```

```
  --  registered versions of the incoming pixels
  signal shpixel_r : pixel_t;
  signal s0pixel_r : pixel_t;
  signal s1pixel_r : pixel_t;
  signal s2pixel_r : pixel_t;
  signal s3pixel_r : pixel_t;

  --  registered reconfigure signals
  signal reconfig_r  : std_logic;
  signal func_r      : func_t;
  signal loadaddr_r  : loadaddr_t;
  signal loaddata_r  : loaddata_t;
  signal clearaddr_r : std_logic;


begin


  --  EAB address counter and other registered statements
  process (clk)
  begin
    if clk'event and clk = '1' then
      shpixel_r   <= shpixel;
      s0pixel_r   <= s0pixel;
      s1pixel_r   <= s1pixel;
      s2pixel_r   <= s2pixel;
      s3pixel_r   <= s3pixel;
      reconfig_r  <= reconfig;
      func_r      <= func;
      loadaddr_r  <= loadaddr;
      loaddata_r  <= loaddata;
      clearaddr_r <= clearaddr;
    end if;
  end process;


  -------------------------------------------------------------------------------
  --  Component Instantiations
  -------------------------------------------------------------------------------
  interface : guts
    port map(clk => clk,
             globalrst  => globalrst,
             clear      => clear,
             shpixel_r  => shpixel_r,
             s0pixel_r  => s0pixel_r,
             s1pixel_r  => s1pixel_r,
             s2pixel_r  => s2pixel_r,
             s3pixel_r  => s3pixel_r,
             reconfig_r => reconfig_r,
             func_r     => func_r,
             loadaddr_r => loadaddr_r,
             loaddata_r => loaddata_r,
             clearaddr_r => clearaddr_r,

             shaddr    => shaddr,
             s0addr    => s0addr,
             s1addr    => s1addr,
             s2addr    => s2addr,
             s3addr    => s3addr,
             newhit    => newhit,
             hits      => hits
             );

end basic;
```

B.1.3 guts

```
--------------------------------------------------------------------------------
--  Takes the pad signals from master.vhd and sends them where they
--  need to go.  Instantiates most everything.
--  Richard Ross
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.surr4_p.all;
use work.addr_p.all;
use work.comps.all;

--------------------------------------------------------------------------------
--  Entity Declaration
--------------------------------------------------------------------------------
entity guts is
  port(
       -- Incoming
       clk        : in std_logic;
       globalrst  : in std_logic;        --  Resets everything, including counts
       clear      : in std_logic;        --  Clears the calculations
       shpixel_r  : in pixel_t;          -- Registered incoming pixels
       s0pixel_r  : in pixel_t;
       s1pixel_r  : in pixel_t;
       s2pixel_r  : in pixel_t;
       s3pixel_r  : in pixel_t;
       reconfig_r : in std_logic;        -- Reconfiguration signals (registered)
       func_r     : in func_t;
       loadaddr_r : in loadaddr_t;
       loaddata_r : in loaddata_t;
       clearaddr_r : in std_logic;
       -- Outgoing
       shaddr     : out extmemaddr_t;     -- Image memory address lines
       s0addr     : out extmemaddr_t;
       s1addr     : out extmemaddr_t;
       s2addr     : out extmemaddr_t;
       s3addr     : out extmemaddr_t;
       newhit     : out std_logic;        -- Signals completion of one
                                          -- iteration
       hits       : out std_logic_vector(3 downto 0)
       );
end guts;



--------------------------------------------------------------------------------
--  architecture declaration
--------------------------------------------------------------------------------
architecture basic of guts is
  --  constants for decoding which constant is being loaded
  constant loadthrmin_c : std_logic_vector(2 downto 0) := "000";
  constant loadthrmax_c : std_logic_vector(2 downto 0) := "001";
  constant loadbrmin_c  : std_logic_vector(2 downto 0) := "010";
  constant loadbias_c   : std_logic_vector(2 downto 0) := "011";
  constant loadsurmin_c : std_logic_vector(2 downto 0) := "100";
  constant loadconst_c  : func_t := "010";
  constant loadoffsets_c  : func_t := "000";

  --  state signals
  signal state       : state_t;

  --  eab interface signals
```

```
  signal maskaddr    : maskaddr_t;
  signal brmask      : eabdata_t;
  signal surmask     : surmask_t;
  signal cache_sel   : std_logic;
  signal cache0_dout : pixel_t;
  signal cache1_dout : pixel_t;
  signal cache0_we   : std_logic;
  signal cache1_we   : std_logic;
  signal brpixel_r : pixel_t;

  --  hitcount signals
  signal brhits    : std_logic_vector(ntemps_c-1 downto 0);
  signal surhits   : std_logic_vector(ntemps_c-1 downto 0);

  --  other correlator interface signals
  signal threshold : thresharray_t;
  signal surpixel  : pixelarray_t;
  signal memenab   : std_logic;
  signal brpixel   : pixel_t;

  --  reconfigure signals (loading flags)
  signal loadconst : std_logic_vector(4 downto 0);
  signal loadoffsets : std_logic;

begin
  --  concurrent statements
  brpixel <= cache0_dout when cache_sel = '0' else
             cache1_dout;
  cache0_we <= cache_sel when
               (state = both) or (state = thresh1) or (state = thresh2)
               else '0';

  cache1_we <= not cache_sel when
               state = both or state = thresh1 or state = thresh2
               else '0';

  memenab     <= '1';
  surpixel(0) <= s0pixel_r;
  surpixel(1) <= s1pixel_r;
  surpixel(2) <= s2pixel_r;
  surpixel(3) <= s3pixel_r;

  process (reconfig_r, func_r)
  begin
    if (reconfig_r = '1' and func_r = loadoffsets_c) then
      loadoffsets <= '1';
    else
      loadoffsets <= '0';
    end if;
  end process;

  process (loadaddr_r, reconfig_r, func_r)
  begin
    if reconfig_r = '1' and func_r = loadconst_c then
      case loadaddr_r(loadaddr_r'left downto loadaddr_r'left-2) is
        when loadthrmin_c =>
          loadconst <= "00001";
        when loadthrmax_c =>
          loadconst <= "00010";
        when loadbrmin_c =>
          loadconst <= "00100";
        when loadbias_c =>
          loadconst <= "01000";
        when loadsurmin_c =>
          loadconst <= "10000";
```

```
      when others =>
        loadconst <= "00000";
    end case;
  else
    loadconst <= "00000";
  end if;
end process;


--  EAB address counter and other registered statements
process (clk)
begin
  if clk'event and clk = '1' then
    brpixel_r <= brpixel;
    loadoffsets <= '0';
    if reconfig_r = '1' then
      if func_r = loadoffsets_c then
        loadoffsets <= '1';
      end if;
      if clearaddr_r = '0' then
        maskaddr <= (others => '0');
      else
        maskaddr <= maskaddr + 1;
      end if;
    else
      case state is
        when reset1 =>
          maskaddr <= (others => '0');
          cache_sel <= '0';
        when init1 =>
          maskaddr <= (others => '0');
          cache_sel <= not cache_sel;
      when others =>
        maskaddr <= maskaddr + 1;
      end case;
    end if;
  end if;
end process;


-------------------------------------------------------------------------------
--  Component Instantiations
-------------------------------------------------------------------------------

--  shapesum and surround correlation
corr : for cnt in 0 to ntemps_c - 1 generate
  shapes : shapesum
    generic map (mynumber => cnt)
    port map (clk       => clk,
              globalrst => globalrst,
              state     => state,
              shpixel   => shpixel_r,
              brpixel   => brpixel_r,
              mask      => brmask(cnt),
              constaddr => loadaddr_r(2 downto 0),
              constflags => loadconst(3 downto 0),
              constdata => loaddata_r,

              threshout => threshold(cnt),
              hit       => brhits(cnt)
              );

  surr : surround
    generic map(number => cnt)
    port map (
```

```
              clk         => clk,
              globalrst => globalrst,
              state       => state,
              pixel       => surpixel,
              mask        => surmask(cnt),
              thresh      => threshold(cnt),
              constaddr   => loadaddr_r(2 downto 0),
              constflag   => loadconst(4),
              constdata   => loaddata_r,

              hit         => surhits(cnt)
              );
end generate;

masker : masks
  port map (
            clk       => clk,
            reconfig  => reconfig_r,
            func      => func_r,
            loadaddr  => loadaddr_r,
            loaddata  => loaddata_r,
            maskaddr  => maskaddr,

            brmask    => brmask,
            surmask   => surmask
            );

--  Hit count calculator
hitcounter :  hitcount
  port map (clk       => clk,
            state     => state,
            brhit     => brhits,
            surhit    => surhits,

            newhit    => newhit,
            hits      => hits
            );

--  off-chip-memory address generator
imaddr :  addrgen
  port map (clk     => clk,
            state   => state,
            loadmem_d => loadoffsets,
            loadaddr => loadaddr_r,
            loaddata => loaddata_r,

            shaddr  => shaddr,
            s0addr => s0addr,
            s1addr => s1addr,
            s2addr => s2addr,
            s3addr => s3addr
            );


--  state machine
state_machine :  s_machine
  port map (clk     => clk,
            globalrst     => globalrst,
            clear => clear,
            addr    => maskaddr,

            state   => state
            );

--  Image caches
```

```
  cache0 : syn_ram_64x8_iror
--pragma translate_off
    generic map ( LPM_FILE  => "" )
--pragma translate_on
    port map (  Data     => shpixel_r,
                Address  => maskaddr,
                WE       => cache0_we,
                Q        => cache0_dout,
                Inclock  => clk,
                Outclock => clk
                );

  cache1 : syn_ram_64x8_iror
--pragma translate_off
    generic map ( LPM_FILE  => "" )
--pragma translate_on
    port map (  Data     => shpixel_r,
                Address  => maskaddr,
                WE       => cache1_we,
                Q        => cache1_dout,
                Inclock  => clk,
                Outclock => clk
                );


end basic;
```

B.1.4 shapesum

```
--------------------------------------------------------------------------------
--  Bright correlation and threshold calculation
--  Richard Ross
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.surr4_p.all;
use work.comps.all;

--------------------------------------------------------------------------------
--  Entity Declaration
--------------------------------------------------------------------------------
entity shapesum is
  generic (mynumber : integer);
  port (clk        : in std_logic;
        globalrst  : in std_logic;
        state      : in state_t;
        shpixel    : in pixel_t;
        brpixel    : in pixel_t;
        mask       : in std_logic;
        constaddr  : in std_logic_vector(2 downto 0);
        constflags : in std_logic_vector(3 downto 0);
        constdata  : in pixel_t;
```

```
        threshout  : out pixel_t;
        hit        : out std_logic
        );
end shapesum;


--------------------------------------------------------------------------------
--  architecture declaration
--------------------------------------------------------------------------------
architecture basic of shapesum is

  --  constant declarations
  constant zeropad : std_logic_vector(accumwidth_c - 9 downto 0) := (others => '0');

  --  type declarations
  subtype accum_t is std_logic_vector (accumwidth_c-1 downto 0);

  --  signals
  signal accum     : accum_t;
  signal sub_out   : pixel_t;
  signal thresh    : pixel_t;
  signal threshold : pixel_t;
  signal brsum     : brsum_t;
  signal mask_d1   : std_logic;
  signal mask_d2   : std_logic;

  --  signals for checking for hits
  signal overthreshmin  : boolean;
  signal underthreshmax : boolean;
  signal overbrightmin  : boolean;
  signal threshmin : pixel_t;
  signal threshmax : pixel_t;
  signal brightmin : brsum_t;

  --  bias to subtract for theshold
  signal bias : pixel_t;

  --  signals for the divider
  signal div_out  : pixel_t;
  signal divcount : unsigned(3 downto 0);
  signal divrst   : std_logic;
  signal onbits   : std_logic_vector(5 downto 0);

  --------------------------------------------------------------------------------
  --  Begin
  --------------------------------------------------------------------------------
begin
  --  Concurrent Statements
  threshout      <= threshold;

  --  Registered signals
  process (clk)
  begin
    if clk'event and clk = '1' then
      mask_d1 <= mask;
      mask_d2 <= mask_d1;
      sub_out <= div_out - bias;
      overthreshmin  <= threshold >= threshmin;
      underthreshmax <= threshold <= threshmax;
      overbrightmin  <= brsum >= brightmin;
      divcount <= divcount + 1;
      divrst <= '0';
      onbits <= onbits + 1;
      if (mask = '1') then
        accum <= accum + (zeropad & shpixel);
```

```
    end if;

    if (mask_d2 = '1' and brpixel >= threshold) then
      brsum <= brsum + 1;
    end if;

    --  Load constants
    if globalrst = '1' then
      threshmin <= (others => '0');
      threshmax <= (others => '0');
      brightmin <= (others => '0');
      bias <= (others => '0');
    else
      if constaddr = mynumber then
        if constflags(0) = '1' then
          threshmin <= constdata;
        end if;
        if constflags(1) = '1' then
          threshmax <= constdata;
        end if;
        if constflags(2) = '1' then
          brightmin <= constdata(brightmin'range);
        end if;
        if constflags(3) = '1' then
          bias <= constdata;
        end if;
      end if;
    end if;

    case state is
      when reset1 =>
        accum       <= (others => '0');
        thresh      <= (others => '0');
        brsum       <= (others => '0');
        hit         <= '0';
      when init1 =>
        threshold   <= thresh;
      when init2 =>
        accum       <= (others => '0');
      when thresh2 =>
        brsum       <= (others => '0');
        onbits      <= (others => '0');
      when wait1 =>
        divrst <= '1';
      when mult3 =>
        if overthreshmin and underthreshmax and overbrightmin then
          hit <= '1';
        else
          hit <= '0';
        end if;
      when sub =>
        thresh      <= sub_out;
      when others =>
        null;
    end case;
  end if;
end process;

-------------------------------------------------------------------------------
--  Component Instantiations
-------------------------------------------------------------------------------
divider : div
  generic map (chunk_n => mynumber)
  port map (
            clk     => clk,
```

```
            reset   => divrst,
            op      => accum,
            divisor => onbits,
            result  => div_out
            );

end basic;
```

B.1.5 div

```
--------------------------------------------------------------------------------
--  Iterative Divider using Patterson and Hennessy p.220
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
--use work.div_p.all;
use work.comps.all;

entity div is
  generic (chunk_n : integer);           -- uniquifies the divider
  port (
        clk    : in std_logic;
        reset  : in std_logic;
        op     : in std_logic_vector(13 downto 0);
        divisor : in std_logic_vector(5 downto 0);
        result : out std_logic_vector(7 downto 0)
        );
end div;

architecture basic of div is
  constant lhalf_c : integer := 6;
  constant rhalf_c : integer := 14;

  signal leftreg : std_logic_vector(lhalf_c - 1 downto 0);
  signal rightreg : std_logic_vector(rhalf_c - 1 downto 0);
  signal subresult : std_logic_vector(lhalf_c - 1 downto 0);

begin
  result <= rightreg(result'range);
  subresult <= leftreg - divisor;
  process (clk)
  begin
    if clk'event and clk = '1' then
      if reset = '1' then
        rightreg <= op;
        leftreg <= (others => '0');
      else
        rightreg(rightreg'left downto 1) <= rightreg(rightreg'left-1 downto 0);
        if (subresult(subresult'left) = '1') then
          rightreg(0) <= '0';              --  negative, restore
          leftreg <= leftreg(leftreg'left-1 downto 0) & rightreg(rightreg'left);
        else
          rightreg(0) <= '1';              --  positive
          leftreg <= subresult(subresult'left-1 downto 0) & rightreg(rightreg'left);
        end if;
      end if;

    end if;
  end process;
end basic;
```

B.1.6 surroundsum

```
--------------------------------------------------------------------------------
--  Surround correlation
--  Richard Ross
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.surr4_p.all;
use work.comps.all;


--------------------------------------------------------------------------------
--  Entity Declaration
--------------------------------------------------------------------------------
entity surround is
  generic (number : integer);
  port (clk       : in std_logic;
        globalrst : in std_logic;
        state     : in state_t;
        pixel     : in pixelarray_t;
        mask      : in std_logic_vector(nsurr_c - 1 downto 0);
        thresh    : in pixel_t;
        constaddr : in std_logic_vector(2 downto 0);
        constflag : in std_logic;
        constdata : in pixel_t;

        hit       : out std_logic
        );
end surround;


--------------------------------------------------------------------------------
--  architecture declaration
--------------------------------------------------------------------------------
architecture basic of surround is

  --  signal declarations
  signal sum            : sursum_t;
  signal intersum0      : sursum4_t;
  signal intersum1      : sursum4_t;
  signal sursum1        : sursum4_t;
  signal oversurmin     : boolean;
  signal surrmin : sursum4_t;

  --------------------------------------------------------------------------------
  --  Begin
  --------------------------------------------------------------------------------
begin
  --  Registered signals
  process (clk)
  begin
    if clk'event and clk = '1' then
      oversurmin <= sursum1 >= surrmin;

      for region in 0 to nsurr_c - 1 loop
        if (mask(region) = '1') and
          (pixel(region) < thresh) then
          sum(region) <= sum(region) + 1;
        end if;
      end loop;

      intersum0 <= ("00" & sum(0)) + ("00" & sum(1));
```

```
      intersum1 <= ("00" & sum(2)) + ("00" & sum(3));

      sursum1  <= intersum0 + intersum1;

      --  Load constants
      if globalrst = '1' then
        surrmin <= (others => '0');
      else
        if constaddr = number and constflag = '1' then
            surrmin <= constdata;
        end if;
      end if;
      case state is

        when reset1 =>
          intersum0   <= (others => '0');
          intersum1   <= (others => '0');
          sursum1    <= (others => '0');
          for region in 0 to nsurr_c - 1 loop
            sum(region) <= (others => '0');
          end loop;

        when init2 =>
          for region in 0 to nsurr_c - 1 loop
            sum(region) <= (others => '0');
          end loop;

        when thresh2 =>
          if oversurmin then
            hit <= '1';
          else
            hit <= '0';
          end if;

        when others =>
          null;
      end case;
    end if;
  end process;

end basic;
```

B.1.7 s_machine

```
--------------------------------------------------------------------------------
--  State Machine
--  Richard Ross
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.surr4_p.all;


--------------------------------------------------------------------------------
--  Entity Declaration
--------------------------------------------------------------------------------
entity s_machine is
  port(clk       : in std_logic;
       globalrst : in std_logic;
       clear     : in std_logic;
       addr      : in maskaddr_t;
```

```
       state     : out state_t
       );
end s_machine;



--------------------------------------------------------------------------------
--  architecture declaration
--------------------------------------------------------------------------------
architecture basic of s_machine is
  signal c_state, n_state : state_t;
  signal divcount : unsigned(3 downto 0);
  constant divmax : unsigned(3 downto 0) := "1110";

begin
  state <= c_state;

  --  Next state calculation
  process (globalrst, clear, addr, c_state, divcount)
  begin
    if (globalrst = '1' or clear = '1') then
      n_state <= reset1;
    else
      case c_state is
        when reset1  => n_state <= reset2;
        when reset2  => n_state <= init1;
        when init1   => n_state <= init2;
        when init2   => n_state <= thresh1;
        when thresh1 => n_state <= thresh2;
        when thresh2 => n_state <= both;
        when both   =>
          if addr(brbits_c'range) = brbits_c then
            n_state <= wait1;
          else
            n_state <= both;
          end if;
        when wait1  => n_state <= mult1;
        when mult1  => n_state <= mult2;
        when mult2  =>
          if divcount = divmax then
            n_state <= mult3;
          else
            n_state <= mult2;
          end if;
        when mult3  => n_state <= sub;
        when sub    => n_state <= surronly;
        when surronly =>
          if addr(surbits_c'range) = (surbits_c) then
            n_state <= wait2;
          else
            n_state <= surronly;
          end if;
        when wait2 => n_state <= init1;
        when others => n_state <= reset1;
      end case;
    end if;
  end process;

  --  Registered statements
  process (clk)
  begin
    if clk'event and clk = '1' then
      c_state <= n_state;
      divcount <= divcount + 1;
      if c_state = mult1 then
```

```
        divcount <= (others => '0');
      end if;
    end if;
  end process;

end basic;
```

B.1.8 masks

```
--------------------------------------------------------------------------------
--  Controls the EABs that hold the template masks.
--  Richard Ross
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.surr4_p.all;
use work.comps.all;

--------------------------------------------------------------------------------
--  Entity Declaration
--------------------------------------------------------------------------------
entity masks is
  port(clk       : in std_logic;
       reconfig  : in std_logic;
       func      : in func_t;
       loadaddr  : in loadaddr_t;
       loaddata  : in loaddata_t;
       maskaddr  : in maskaddr_t;

       brmask    : out eabdata_t;
       surmask   : out surmask_t
       );
end masks;



--------------------------------------------------------------------------------
--  architecture declaration
--------------------------------------------------------------------------------
architecture basic of masks is
  constant brfunc_c    : func_t := "001";
  constant surr0func_c : func_t := "100";
  constant surr1func_c : func_t := "101";
  constant surr2func_c : func_t := "110";
  constant surr3func_c : func_t := "111";

  signal addr : maskaddr_t;
  signal surrdata     : eabdatarray_t;
  signal brwe : std_logic;
  signal surr0we : std_logic;
  signal surr1we : std_logic;
  signal surr2we : std_logic;
  signal surr3we : std_logic;

begin
  brwe <= '1' when reconfig = '1' and func = brfunc_c else
          '0';
  surr0we <= '1' when reconfig = '1' and func = surr0func_c else
             '0';
  surr1we <= '1' when reconfig = '1' and func = surr1func_c else
             '0';
  surr2we <= '1' when reconfig = '1' and func = surr2func_c else
```

```
           '0';
  surr3we <= '1' when reconfig = '1' and func = surr3func_c else
           '0';
  addr <= loadaddr(addr'range) when reconfig = '1' else
          maskaddr;

  process(surrdata)
  begin
    for template in ntemps_c - 1 downto 0 loop
      for quad in nsurr_c - 1 downto 0 loop
        surmask(template)(quad) <= surrdata(quad)(template);
      end loop;
    end loop;
  end process;

  -------------------------------------------------------------------------
  --  Component Instantiations
  -------------------------------------------------------------------------
  --  shapesum mask EAB
  brmaskmem : syn_ram_64x8_iror
--pragma translate_off
    generic map( LPM_FILE => "")
--pragma translate_on
    port map(
             Data     => loaddata,
             Address  => addr,
             WE       => brwe,
             Q        => brmask,
             Inclock  => clk,
             Outclock => clk
             );

  --  surround0 mask EAB
  surrmask0 : syn_ram_64x8_iror
--pragma translate_off
    generic map ( LPM_FILE => "")
--pragma translate_on
    port map(
             Data     => loaddata,
             Address  => addr,
             WE       => surr0we,
             Q        => surrdata(0),
             Inclock  => clk,
             Outclock => clk
             );

  --  surround1 mask EAB
  surrmask1 : syn_ram_64x8_iror
--pragma translate_off
    generic map ( LPM_FILE => "")
--pragma translate_on
    port map (
              Data     => loaddata,
              Address  => addr,
              WE       => surr1we,
              Q        => surrdata(1),
              Inclock  => clk,
              Outclock => clk
              );

  --  surround2 mask EAB
  surrmask2 : syn_ram_64x8_iror
--pragma translate_off
    generic map ( LPM_FILE => "")
--pragma translate_on
```

```
    port map (
              Data     => loaddata,
              Address  => addr,
              WE       => surr2we,
              Q        => surrdata(2),
              Inclock  => clk,
              Outclock => clk
              );

  --  surround3 mask EAB
  surrmask3 : syn_ram_64x8_iror
--pragma translate_off
    generic map ( LPM_FILE => "")
--pragma translate_on
    port map (
              Data     => loaddata,
              Address  => addr,
              WE       => surr3we,
              Q        => surrdata(3),
              Inclock  => clk,
              Outclock => clk
              );



end basic;
```

B.1.9 addrgen

```
--------------------------------------------------------------------------------
--  Address generator for off-chip RAM
--  Richard Ross
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.surr4_p.all;
use work.addr_p.all;
use work.comps.all;

--------------------------------------------------------------------------------
--  Entity Declaration
--------------------------------------------------------------------------------
entity addrgen is
  port(clk        : in std_logic;
       state      : in state_t;
       loadmem_d  : in std_logic;
       loadaddr   : in loadaddr_t;
       loaddata   : loaddata_t;

       shaddr   : out extmemaddr_t;
       s0addr   : out extmemaddr_t;
       s1addr   : out extmemaddr_t;
       s2addr   : out extmemaddr_t;
       s3addr   : out extmemaddr_t
       );
end addrgen;



--------------------------------------------------------------------------------
--  architecture declaration
```

```
--------------------------------------------------------------------------------
architecture basic of addrgen is

  signal eabaddr    : std_logic_vector(5 downto 0);
  signal eabaddr_m  : std_logic_vector(5 downto 0);
  signal xinc, yinc : std_logic_vector (3 downto 0);
  signal eabdata    : std_logic_vector(7 downto 0);
  signal xbaseaddr  : std_logic_vector(6 downto 0);
  signal ybaseaddr  : std_logic_vector(6 downto 0);
  signal memenab    : std_logic;
  signal surrcount  : std_logic_vector(5 downto 0);
  signal loadaddr_d : loadaddr_t;
  signal loaddata_d : loaddata_t;

  signal s0,s1,s2,s3 : extmemaddr_t;
  ------------------------------------------------------------------------------
  --  Valid pixel values are needed on all cycles in the "both" and "thresh" states.
  --  This is how things need to be synchronized. Call the cycle when the pixel
  --  is needed time = t.
  --  t-4:  EAB address calculated (cleared or incremented) and registered
  --  t-3:  xinc and yinc read from EAB and registered.  Baseaddr
  --        calculated (cleared, incremented, or nothing) and registered
  --  t-2:  pixel address calculated (baseaddr + inc) and registered
  --  t-1:  pixel read and registered.
  --  t  :  pixel used
  ------------------------------------------------------------------------------
begin

  --  concurrent statements
  memenab  <= '1';
  xinc     <= eabdata(xinc'range);
  yinc     <= eabdata(7 downto 4);
  eabaddr_m <= loadaddr_d(eabaddr_m'range) when loadmem_d = '1' else
               eabaddr;

  --  clocked statements
  process (clk)
  begin
    if clk'event and clk = '1' then
      loadaddr_d <= loadaddr;
      loaddata_d <= loaddata;
      eabaddr   <= eabaddr + 1;
      surrcount <= surrcount + 1;
      shaddr    <= (ybaseaddr + ("000" & yinc)) &
                   (xbaseaddr + ("000" & xinc));
      s0addr    <= (ybaseaddr + ("0000" & surrcount(5 downto 3))) &
                   (xbaseaddr + ("0000" & surrcount(2 downto 0)));
      s1addr    <= (ybaseaddr + ("0000" & surrcount(5 downto 3))) &
                   (xbaseaddr + ("0001" & surrcount(2 downto 0)));
      s2addr    <= (ybaseaddr + ("0001" & surrcount(5 downto 3))) &
                   (xbaseaddr + ("0000" & surrcount(2 downto 0)));
      s3addr    <= (ybaseaddr + ("0001" & surrcount(5 downto 3))) &
                   (xbaseaddr + ("0001" & surrcount(2 downto 0)));
      case state is
        when reset1 =>
          xbaseaddr <= (others => '0');
          ybaseaddr <= (others => '0');
          eabaddr   <= (others => '0');
          surrcount <= (others => '0');

        when reset2 =>
          xbaseaddr <= (others => '0');
          ybaseaddr <= (others => '0');
          surrcount <= (others => '0');
```

```
        when surronly =>
          eabaddr <= (others => '0');

        when wait2 =>
          if xbaseaddr = search_max_c then
            xbaseaddr <= (others => '0');
            ybaseaddr <= ybaseaddr + 1;
          else
            xbaseaddr <= xbaseaddr + 1;
          end if;
          surrcount <= (others => '0');

        when others => NULL;
      end case;
    end if;
  end process;

  ---------------------------------------------------------------------------
  --  Component Instantiations
  ---------------------------------------------------------------------------
  brmap : syn_ram_64x8_iror
--pragma translate_off
    generic map (LPM_FILE => "")
--pragma translate_on
    port map (
              Data   => loaddata_d,
              Address => eabaddr_m,
              WE => loadmem_d,
              Q => eabdata,
              Inclock => clk,
              Outclock => clk
              );

end basic;
```

B.1.10 hitcount

```
---------------------------------------------------------------------------
--  Hitcount calculation
--  Richard Ross
---------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use work.surr4_p.all;
use work.comps.all;


---------------------------------------------------------------------------
--  Entity Declaration
---------------------------------------------------------------------------
entity hitcount is
  port (clk     : in std_logic;
        state   : in state_t;
        brhit   : in std_logic_vector(ntemps_c-1 downto 0);
        surhit  : in std_logic_vector(ntemps_c-1 downto 0);

        newhit  : out std_logic;
        hits    : out std_logic_vector(3 downto 0)
        );
end hitcount;
```

```
--------------------------------------------------------------------------------
--  architecture declaration
--------------------------------------------------------------------------------
architecture basic of hitcount is

  --  type declarations
  subtype decode_t is std_logic_vector(2 downto 0);

  --  signal declarations
  signal bothhit  : std_logic_vector(7 downto 0);
  signal count    : std_logic_vector(3 downto 0);
  signal opleft, opright : decode_t;

  -- Counts the incoming hits from the bright and surround correlations
  function decode (op : std_logic_vector(3 downto 0))
    return decode_t is
  begin
    case op is
      when "0000" => return "000";
      when "0001" => return "001";
      when "0010" => return "001";
      when "0011" => return "010";

      when "0100" => return "001";
      when "0101" => return "010";
      when "0110" => return "010";
      when "0111" => return "011";

      when "1000" => return "001";
      when "1001" => return "010";
      when "1010" => return "010";
      when "1011" => return "011";

      when "1100" => return "010";
      when "1101" => return "011";
      when "1110" => return "011";
      when "1111" => return "111";

      when others => return "---";
    end case;
  end decode;

  --------------------------------------------------------------------------------
  --  Begin
  --------------------------------------------------------------------------------

begin

  --  Registered signals
  process (clk)
  begin
    if clk'event and clk = '1' then
      bothhit <= (others => '0');
      bothhit(brhit'range) <= brhit and surhit;
      opright <= decode(bothhit(3 downto 0));
      opleft  <= decode(bothhit(7 downto 4));
      count   <= ("0" & opleft) + ("0" & opright);

      case state is
        when reset1 =>
          hits <= (others => '0');
          newhit <= '0';

        when wait1 =>
          hits <= count;
```

```
          newhit <= '1';                -- Signals a new result
        when others =>
          newhit <= '0';

      end case;
    end if;
  end process;

end basic;
```

B.2 Package Files

B.2.1 surr4_p

```
--------------------------------------------------------------------------------
--  Package for the whole design.  Contains most type declarations and
--  constants.
--  Richard Ross
--------------------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

package surr4_p is
  --  Make design as paramaterized as possible.  Put constants and types in
  --  one file so they are easy to change.

  constant ntemps_c             : integer := 8;     --  #of templates
  constant nsurr_c              : integer := 4;     --  #of regions for surround
  constant brsumwidth_c         : integer := 6;     --  width of the bright sum in bits
  constant sursumwidth_c        : integer := 6;     --  width of one surround sum
  constant sursum4width_c       : integer := sursumwidth_c+2; --  with of the total surround sum
  constant maskaddrwidth_c      : integer := sursumwidth_c;
  constant accumwidth_c         : integer := 14;
  constant eabdatawidth         : integer := 8;
  constant funcwidth_c          : integer := 3;
  constant loadaddrwidth_c      : integer := 8;
  constant loaddatawidth_c      : integer := 8;

  subtype brsum_t    is std_logic_vector(brsumwidth_c - 1    downto 0); --  bright sum type
  type    sursum_t   is array (nsurr_c-1 downto 0) of std_logic_vector(sursumwidth_c - 1 downto 0);
  subtype sursum4_t  is std_logic_vector(sursum4width_c - 1  downto 0);
  subtype maskaddr_t is std_logic_vector(maskaddrwidth_c - 1 downto 0);
  subtype eabdata_t  is std_logic_vector(eabdatawidth-1      downto 0);
  subtype loaddata_t is std_logic_vector(loaddatawidth_c-1  downto 0);
  subtype loadaddr_t is std_logic_vector(loadaddrwidth_c-1   downto 0);
  subtype func_t     is std_logic_vector(funcwidth_c-1       downto 0);

  constant brbits_c  : brsum_t :=  "011000"; --  #bits on - 1 in combined templates
  constant surbits_c : std_logic_vector(sursumwidth_c - 1 downto 0) := "111110";

  subtype pixel_t       is std_logic_vector(7 downto 0);
  type pixelarray_t     is array (nsurr_c-1 downto 0) of pixel_t;
  type brsumarray_t     is array (ntemps_c-1 downto 0) of brsum_t;
  type sursum4array_t   is array (ntemps_c-1 downto 0) of sursum4_t;
  type sursuminter_t    is array (ntemps_c-1 downto 0) of sursum_t;--  dummy type
  type sursumarray_t    is array (nsurr_c-1 downto 0) of sursuminter_t;
  type eabdatarray_t    is array (nsurr_c - 1 downto 0) of eabdata_t;
  type thresharray_t    is array (ntemps_c-1 downto 0) of pixel_t;
  type surmask_t        is array (ntemps_c-1 downto 0)
    of std_logic_vector(nsurr_c - 1 downto 0);
```

```
  --  State machine encoding
  type state_t is (reset1, reset2, init1, init2, thresh1, thresh2, both, wait1,
                   mult1, mult2, mult3, sub, surronly, wait2);
--  attribute enum_encoding : string;
--  attribute enum_encoding of state_t : type is
--    (
--     "10000000000000 " & "01000000000000 " & "00100000000000 " & "00010000000000 " &
--     "00001000000000 " & "00000100000000 " & "00000010000000 " & "00000001000000 " &
--     "00000000100000 " & "00000000010000 " & "00000000001000 " & "00000000000100 " &
--     "00000000000010 " & "00000000000001 "
--     );


end surr4_p;

package body surr4_p is

end surr4_p;
```

B.2.2 addr_p

```
--------------------------------------------------------------------------------
--  Package for the address generator
--  Richard Ross
--------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.surr4_p.all;

package addr_p is
  constant extmemaddrw_c : integer := 14;
  constant search_max_c : std_logic_vector(6 downto 0) := "1110000";
  subtype extmemaddr_t is std_logic_vector (extmemaddrw_c-1 downto 0);

end addr_p;

package body addr_p is

end addr_p;
```

B.2.3 comps

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164.all;
```

```
use ieee.std_logic_unsigned.all;
use work.surr4_p.all;
use work.addr_p.all;

package comps is

 component addrgen
  port(clk        : in std_logic;
       state      : in state_t;
       loadmem_d  : in std_logic;
       loadaddr   : in loadaddr_t;
       loaddata   : loaddata_t;

       shaddr   : out extmemaddr_t;
       s0addr   : out extmemaddr_t;
       s1addr   : out extmemaddr_t;
       s2addr   : out extmemaddr_t;
       s3addr   : out extmemaddr_t
       );
 end component;

 component div
  generic (chunk_n : integer);           -- uniquifies the divider
  port (
        clk    : in std_logic;
        reset  : in std_logic;
        op     : in std_logic_vector(13 downto 0);
        divisor : in std_logic_vector(5 downto 0);
        result : out std_logic_vector(7 downto 0)
        );
 end component;

 component guts
  port(
       -- Incoming
       clk        : in std_logic;
       globalrst  : in std_logic;        --  Resets everything, including counts
       clear      : in std_logic;        --  Clears the calculations
       shpixel_r  : in pixel_t;          -- Registered incoming pixels
       s0pixel_r  : in pixel_t;
       s1pixel_r  : in pixel_t;
       s2pixel_r  : in pixel_t;
       s3pixel_r  : in pixel_t;
       reconfig_r : in std_logic;        -- Reconfiguration signals (registered)
       func_r     : in func_t;
       loadaddr_r : in loadaddr_t;
       loaddata_r : in loaddata_t;
       clearaddr_r : in std_logic;
       -- Outgoing
       shaddr    : out extmemaddr_t;     -- Image memory address lines
       s0addr    : out extmemaddr_t;
       s1addr    : out extmemaddr_t;
       s2addr    : out extmemaddr_t;
       s3addr    : out extmemaddr_t;
       newhit    : out std_logic;        -- Signals completion of one
                                         -- iteration
       hits      : out std_logic_vector(3 downto 0)
       );
 end component;

 component hitcount
  port (clk     : in std_logic;
        state   : in state_t;
        brhit   : in std_logic_vector(ntemps_c-1 downto 0);
        surhit  : in std_logic_vector(ntemps_c-1 downto 0);
```

```
       newhit  : out std_logic;
       hits    : out std_logic_vector(3 downto 0)
       );
end component;

component masks
 port(clk       : in std_logic;
      reconfig  : in std_logic;
      func      : in func_t;
      loadaddr  : in loadaddr_t;
      loaddata  : in loaddata_t;
      maskaddr  : in maskaddr_t;

      brmask    : out eabdata_t;
      surmask   : out surmask_t
      );
end component;

component master
 port(clk       : in std_logic;
      globalrst : in std_logic;     --  Resets everything, including counts
      clear     : in std_logic;             --  Clears the calculations
      shpixel   : in pixel_t;
      s0pixel   : in pixel_t;
      s1pixel   : in pixel_t;
      s2pixel   : in pixel_t;
      s3pixel   : in pixel_t;
      reconfig  : in std_logic;
      func      : in func_t;
      loadaddr  : in loadaddr_t;
      loaddata  : in loaddata_t;
      clearaddr : in std_logic;

      shaddr    : out extmemaddr_t;
      s0addr    : out extmemaddr_t;
      s1addr    : out extmemaddr_t;
      s2addr    : out extmemaddr_t;
      s3addr    : out extmemaddr_t;
      newhit    : out std_logic;
      hits      : out std_logic_vector(3 downto 0)
      );
end component;

component s_machine
 port(clk       : in std_logic;
      globalrst : in std_logic;
      clear     : in std_logic;
      addr      : in maskaddr_t;

      state     : out state_t
      );
end component;

component shapesum
 generic (mynumber : integer);
 port (clk        : in std_logic;
       globalrst  : in std_logic;
       state      : in state_t;
       shpixel    : in pixel_t;
       brpixel    : in pixel_t;
       mask       : in std_logic;
       constaddr  : in std_logic_vector(2 downto 0);
       constflags : in std_logic_vector(3 downto 0);
       constdata  : in pixel_t;
```

```
        threshout  : out pixel_t;
        hit        : out std_logic
        );
 end component;

 component surround
  generic (number : integer);
  port (clk       : in std_logic;
        globalrst : in std_logic;
        state     : in state_t;
        pixel     : in pixelarray_t;
        mask      : in std_logic_vector(nsurr_c - 1 downto 0);
        thresh    : in pixel_t;
        constaddr : in std_logic_vector(2 downto 0);
        constflag : in std_logic;
        constdata : in pixel_t;

        hit       : out std_logic
        );
 end component;

component syn_ram_64x8_iror
--pragma translate_off
     generic ( LPM_FILE : string );
--pragma translate_on
     port (  Data    : in std_logic_vector(7 downto 0);
             Address : in std_logic_vector(5 downto 0);
             WE : in std_logic;
             Q  : out std_logic_vector(7 downto 0);
             Inclock  : in std_logic;
             Outclock : in std_logic
     );
end component;
 component tb_addrgen
 end component;

 component tb_div
 end component;

 component tb_hitcount
 end component;

 component tb_master
 end component;

 component tb_testmult
 end component;

 component tb_shapesum
 end component;

 component tb_surround
 end component;


END comps;
```

Appendix C

C++ SOURCE CODE

This appendix contains the C++ code that was used to generate synthetic templates. The algorithm is explained in Appendix A.

C.1 Program Code

C.1.1 prog.C

```
#include <time.h>
#include <stdlib.h>
#include "matrix.h"

#define MINBITS 4
#define MAXBITS 40

void MatStuff(int, double, double, double);

int main(int argc, char *argv[])
{
   int ntemps, xdim, ydim, iterations, i, pbmcount=0, graphcount=0, ave, avecount;
   double params[3][3], a, b1, b2;
   time_t seed;
   group *temps;
   FILE *paramfile;
   char filename[32], instring[64], idstring[32];

   //check usage
   if (argc < 3) {
      fprintf(stderr, "\nUsage: %s <ntemps> <paramfile> \n\n", argv[0]);
      return 1;
   }
   // Seed random number generator
   srand48(time(&seed));
   sscanf(argv[1], "%d", &ntemps);
   // open paramter file
   if ((paramfile = fopen(argv[2], "r")) == 0) {
      fprintf(stderr, "\nCouldn't open parameter file \"%s\"\n\n", argv[2]);
      return 1;
   }
   // Read paramters
   fgets(instring, 256, paramfile);
   sscanf(instring, " %d %d", &ydim, &xdim);
   fgets(instring, 256, paramfile);
   sscanf(instring, " %d", &iterations);
   for (i=0; i<3; ++i) {
      fgets(instring, 256, paramfile);
      sscanf(instring, " %lf %lf %lf", &params[i][0], &params[i][1], &params[i][2]);
   }
   fclose(paramfile);

   // Generate templates
   temps = new group(ydim, xdim, ntemps);
```

```
   printf("\n%% MATLAB output.  Place in a .m file and run from MATLAB.\n");
   printf("x = %d:%d;\n", MINBITS, MAXBITS);
   for (b2=params[2][0]; b2 <= params[2][2]; b2 += params[2][1]) {
      for (b1=params[1][0]; b1 <= params[1][2]; b1 += params[1][1]) {
//  for (a=params[0][0]; a <= params[0][2]; a += params[0][1]) {
 a = -2.0 * b1;
 if ((graphcount%12) == 0) {
    printf("figure(%d)\n", graphcount/12 + 1);
 }
 ++graphcount;
 printf("\ny%d = [", graphcount);
 for (i=MINBITS; i<=MAXBITS; ++i) {
    ave = 0;
    for (avecount = 0; avecount < 3; ++avecount) {
       //   printf("trial %d ", avecount);
       temps->MakeTemps(iterations, i, a, b1, b2);
       ave += temps->CountBits();
    }
    ave /= 3;
    if (i==12) {
       sprintf(filename, "group%d.pbm", pbmcount);
       sprintf(idstring, "onbits = %d \ta = %0.1f \tb1 = %0.1f \tb2 = %0.1f",
       i, a, b1, b2);
       temps->MakePBM(filename, idstring);
       ++pbmcount;
    }
    printf("%d ", ave);
 }
 MatStuff(graphcount, a, b1, b2);
 //  }
      }
   }
   if (graphcount%12)
printf("\norient tall;\npreprint;\nprint -deps plot%d\n", graphcount / 12);
   return 0;
}


void MatStuff(int num, double a, double b1, double b2)
{
   printf("];\n");
   printf("subplot(4,3,%d); plot(x,y%d);\nhold;\n", (num-1)%12+1, num);
   printf("%s%s",
  "plot([0 30], [50 50]);\n",
  "axis([0 30 0 100]);\n");
   printf("string = '\\link \\small \\alpha = %0.1f \\n \\beta_{1,*} = %0.1f \\n \\beta_{2,*} = %0.1f';\n",
  a, b1, b2);
   printf("if y%d(15) > 50\n\tt1 = text(15, 30, string);\n", num);
   printf("else\n\tt1 = text(1, 85, string);\nend;\nlink(t1);");
   if ((num%12) == 0)
printf("\norient tall;\npreprint;\nprint -deps plot%d.eps\n", num / 12);
}
```

C.1.2 matrix.C

```
#include <stdlib.h>// for drand48()
#include <math.h>// for exp()
#include "matrix.h"
```

```
////////////////////////////////////////////////////////////////////////////
// matrix class
////////////////////////////////////////////////////////////////////////////
matrix::matrix(int y, int x)
{
   xdim = x;
   ydim = y;
   data =  new element_t[xdim*ydim];
   for (x=0; x<xdim; ++x)
for(y=0; y<ydim; ++y)
     SetEl(y, x, (element_t)0);
}

matrix::matrix(char filename[])
{
   int x,y;
   FILE *infile;
   char magic[2], inchar;

   if ((infile = fopen(filename, "r"))==NULL) {
      fprintf(stderr, "Couldn't open %s\n", filename);
      xdim = 0; ydim = 0;
      data = NULL;
      return;
   }
   fscanf(infile, " %c%c", magic, magic+1);
   if (magic[0]=='P' && magic[1]=='1') {
      fscanf(infile, " %d %d", &xdim, &ydim);
      data = new element_t[xdim*ydim];
      for (y=0; y<ydim; ++y) {
 for (x=0; x<xdim; ++x) {
    fscanf(infile, " %c", &inchar);
    SetEl(y,x,(inchar=='0') ? 0 : 1);
 }
      }
   }
   else {
      fprintf(stderr, "Not an ASCII PBM file\n");
   }
}



matrix::~matrix(void)
{
   delete data;
}


void matrix::SetEl(int y, int x, element_t d)
{
   if ((y>=0) && (y<ydim) && (x>=0) && (x<xdim)) {
      data[y*xdim+x] = d;
   }
   else {
      fprintf(stderr, "\nERROR!!! SetEl accessed (%d,%d).  Max (%d,%d)\n\n",
      y, x, ydim, xdim);
      exit(1);
   }
}


element_t matrix::GetEl(int y, int x)
{
   if ((y>=0) && (y<ydim) && (x>=0) && (x<xdim))
```

```
return data[y*xdim+x];
   else {
      fprintf(stderr, "\nERROR!!! GetEl accessed (%d,%d).  Max (%d,%d)\n\n",
      y, x, ydim, xdim);
      exit(1);
      return (element_t)0;
   }
}


void matrix::Clear(void)
{
   int x,y;

   for (x=0; x<xdim; ++x)
for(y=0; y<ydim; ++y)
     SetEl(y, x, (element_t)0);
}


void matrix::Scale(double factor)
{
   int x, y;

   for (y=0; y<ydim; ++y) {
     for (x=0; x<xdim; ++x) {
       SetEl(y,x,(element_t)((double)GetEl(y,x) * factor));
     }
   }
}

void matrix::Scale(int factor)
{
   int x, y;

   for (y=0; y<ydim; ++y) {
      for (x=0; x<xdim; ++x) {
 SetEl(y,x,GetEl(y,x) / factor);
      }
   }
}



void matrix::Show(void)
{
   int x,y, index=0;

   for (y=0; y<ydim; ++y) {
      for (x=0; x<xdim; ++x) {
 printf("%d ", data[index++]);
      }
      putchar('\n');
   }
   putchar('\n');
}

void matrix::Show(FILE *outfile)
{
   int x,y, index=0;

   for (y=0; y<ydim; ++y) {
      for (x=0; x<xdim; ++x) {
 fprintf(outfile, "%d ", data[index++]);
      }
```

```
      putc('\n', outfile);
   }
   putc('\n', outfile);
}

void matrix::MakePBM(char *filename, int factor, int binary, char comment[])
{
   FILE *outfile;
   int x,y, m,n, index=0;
   int xdim2 = xdim*factor, ydim2 = ydim*factor;
   unsigned char outbyte=0;

   if ((outfile = fopen(filename, "w")) == 0) {
      fprintf(stderr, "\n\nERROR:  Couldn't open %s to write pbm file.\n\n", filename);
      return;
   }
   if (binary) {
      fprintf(outfile, "P4\n#%s\n%d %d\n", comment, xdim2, ydim2);
      for (y=0; y<ydim; ++y) {
 for (m=0; m<factor; ++m) {
   for (x=0; x<xdim; ++x) {
      for (n=0; n<factor; ++n) {
  outbyte |= GetEl(y,x)? 1:0;
  if (index == 7) {
     index=0;
     putc(outbyte, outfile);
     outbyte = 0;
  }
  else {
     outbyte <<= 1;
     ++index;
  }
       }
    }
    if (index) {
       outbyte <<= (7-index);
       putc(outbyte, outfile);
       index=0;
       outbyte = 0;
    }
 }
      }
   }
   else {
      // print header
      fprintf(outfile, "P1\n#%s\n%d %d\n", comment, xdim2, ydim2);
      for (y=0; y<ydim; ++y) {
 for (m=0; m<factor; ++m) {
    for (x=0; x<xdim; ++x)
 for (n=0; n<factor; ++n)
      fprintf(outfile, "%d ", GetEl(y,x));
    putc('\n', outfile);
 }
      }
   }
   fclose(outfile);
}


////////////////////////////////////////////////////////////////////////
// btemplate class
////////////////////////////////////////////////////////////////////////
btemplate::btemplate(int y, int x) : matrix(y, x)
{
   dim = y*x;
```

```
   onbits = 0;
}

btemplate::btemplate(char filename[]) : matrix(filename)
{
   int x,y,xdim=GetXsize(), ydim=GetYsize();

   dim = xdim*ydim;
   for (y=0; y<ydim; ++y) {
      for (x=0; x<xdim; ++x) {
 if (GetEl(y,x))
      ++onbits;
      }
   }
}


btemplate::~btemplate(void)
{

}


void btemplate::Initialize(int bits, int o, double params[5][2])
{
   int i, m, n;

   order = o;
   nchanges = 0;
   Clear();
   for (m=0; m < 5; ++m)
for (n=0; n < 2; ++n)
     b[m][n] =  params[m][n];
   for (i=0; i<bits; ++i)
SetEl((int)(GetYsize() * drand48()), (int)(GetXsize() * drand48()), 1);
}

int btemplate::Pick2(int pix[])
{
   int rval1, rval2;

   rval1 = (int)(drand48() * dim);
   for (rval2 = rval1; rval2 == rval1; rval2 = (int)(drand48() * dim));
   pix[0] = rval1 / GetXsize();
   pix[1] = rval1 % GetXsize();
   pix[2] = rval2 / GetXsize();
   pix[3] = rval2 % GetXsize();
   return (GetEl(pix[0], pix[1])==0) && (GetEl(pix[2], pix[3])==1) ||
(GetEl(pix[0], pix[1])==1 && GetEl(pix[2], pix[3])==0);
//   return 1;
}

void btemplate::Switch(int pix[])
{
   element_t temp = GetEl(pix[0], pix[1]);
   SetEl(pix[0], pix[1], GetEl(pix[2], pix[3]));
   SetEl(pix[2], pix[3], temp);
   ++nchanges;
}

double btemplate::Ratio(int pix[])
{
   double px, py, T1, T2;
//  double d1, d2, dyfact=1.0;
//   double xcenter = ((double)GetXsize()-1.0)/2.0, ycenter = ((double)GetYsize()-1.0)/2.0;
```

```
   int pix1 = GetEl(pix[0], pix[1]), pix2 = GetEl(pix[2], pix[3]);

   T1 = CalcT(pix);
   T2 = CalcT(pix+2);
   px = exp(pix1 * T1) / ( 1 + exp(T1)) * exp(pix2 * T2) / ( 1 + exp(T2));
   py = exp(pix1 * T2) / ( 1 + exp(T2)) * exp(pix2 * T1) / ( 1 + exp(T1));
   return py/px;
}



double btemplate::CalcT(int pix[])
{
   double T = 0.0;
   element_t val[4];

   val[0] = InRange(pix[0], pix[1]-1) ? GetEl(pix[0], pix[1] - 1): (element_t)0 ;
   val[1] = InRange(pix[0], pix[1]+1) ? GetEl(pix[0], pix[1] + 1): (element_t)0 ;
   val[2] = InRange(pix[0]-1, pix[1]) ? GetEl(pix[0]-1, pix[1]): (element_t)0 ;
   val[3] = InRange(pix[0]+1, pix[1]) ? GetEl(pix[0]+1, pix[1]): (element_t)0 ;
   T = b[0][0] + b[1][0] * (val[0]+val[1]) + b[1][1] * (val[2]+val[3]);
   if (order > 1) {
      val[0] = InRange(pix[0]-1, pix[1]-1) ? GetEl(pix[0]-1, pix[1]-1): (element_t)0 ;
      val[1] = InRange(pix[0]+1, pix[1]+1) ? GetEl(pix[0]+1, pix[1]+1): (element_t)0 ;
      val[2] = InRange(pix[0]-1, pix[1]+1) ? GetEl(pix[0]-1, pix[1]+1): (element_t)0 ;
      val[3] = InRange(pix[0]+1, pix[1]-1) ? GetEl(pix[0]+1, pix[1]-1): (element_t)0 ;
      T += b[2][0] * (val[0]+val[1]) + b[2][1] * (val[2]+val[3]);
   }
   return T;
}


void btemplate::Generate(int bits, int o, double params[5][2], int maxits)
{
   int pix[4], iterations, i, nswitches=0, rswitches=0;
   int x, y, ydim, xdim, maxattempts;
   double ratio, u;

//   printf("btemplate::Generate this=0x%x\n", this);
   ydim = GetYsize();
   xdim = GetXsize();
   maxattempts=xdim*ydim;
   Initialize(bits, o, params);
   for (iterations=0; iterations<maxits; ++iterations) {
      i=0;
      while (i < maxattempts) {
 if (Pick2(pix)) {
    ++i;
    ratio = Ratio(pix);
    if (ratio >= 1) {
       ++nswitches;
       Switch(pix);
    }
    else {
       u = drand48();
       if (ratio > u) {
  ++rswitches;
  Switch(pix);
       }
    }
 }
      }
   }

   // Count the number of on bits
```

```
   onbits=0;
   for (y=0; y<ydim; ++y)
for (x=0; x<xdim; ++x)
     if (GetEl(y,x))
  ++onbits;
   // Center the template
   Center();
   // Print statistics
//    printf("Attempted   Sucess   %% Success  Calculated   Random  %% Random  Onbits\n");
//    printf("%9d%9d%9d%12d%9d%9d%9d\n",
//    attempts, nswitches+rswitches, ((rswitches+nswitches)*100)/attempts,
//    nswitches, rswitches, (rswitches*100)/(nswitches+rswitches), onbits);
}

void btemplate::Center(void)
{
   int x,y, xdim=GetXsize(), ydim=GetYsize(), xcent=0, ycent=0, newx, newy;
   matrix buffer(ydim, xdim);

   buffer.Clear();
   for (y=0; y<ydim; ++y)
for (x=0; x<xdim; ++x)
     if (GetEl(y,x)) {
ycent += y;
xcent += x;
buffer.SetEl(y,x,1);
     }
   ycent = ydim/2 - ycent/onbits;
   xcent = xdim/2 - xcent/onbits;
   Clear();
   for (y=0; y<ydim; ++y)
for (x=0; x<xdim; ++x)
     if (buffer.GetEl(y,x)) {
if ((newx=(x+xcent)%xdim)<0)
     newx += xdim;
if ((newy=(y+ycent)%ydim)<0)
     newy += ydim;
SetEl(newy, newx, 1);
     }
}


// void btemplate::PrintLine(FILE *outfile)
// {
//    int x, xdim = GetXsize();

//    for (x=0; x<xdim; ++x)
//  fprintf(outfile, "%d ", GetEl(y,x));
// }



////////////////////////////////////////////////////////////////////////////
// group class
////////////////////////////////////////////////////////////////////////////
group::group(int y, int x, int n)
{
   int i,j;

   ntemps = n;
   xdim = x;
   ydim = y;
   master = NULL;
   templates = new (btemplate*)[ntemps];
   for (i=0; i<ntemps; ++i) {
```

```
      templates[i] = new btemplate(ydim, xdim);
   }
   for (i=0; i<5; ++i)
for (j=0; j<2; ++j)
     params[i][j] = 0.0;
}

group::~group(void)
{
   int i;

   for (i=0; i<ntemps; ++i)
if (templates[i])
     delete templates[i];
   delete templates;
}



void group::MakeTemps(int its, int onbits_i, double a, double b1, double b2)
{
   int i;

   onbits = onbits_i;
   order = 2;
   iterations = its;
   params[0][0] = a;
   params[1][0] = b1;
   params[1][1] = b1;
   params[2][0] = b2;
   params[2][1] = b2;
   for (i=0; i<ntemps; ++i) {
      templates[i]->Generate(onbits, order, params, iterations);
//      sprintf(filename, "template%d.pbm", i);
//      templates[i]->MakePBM(filename);
   }
}

void group::MakePBM(char filename[], char comment[])
{
   int i, j, x, y, column=0, row=0, border=1, thisrow;
   int xsize = 4*(xdim+border)+border, ysize = ((ntemps-1)/4+1)*(ydim+border)+border;
   matrix data(ysize, xsize);
//    char file2[32];

   for (y=0; y<border; ++y) {
      column=0;
      for (x=0; x<xsize; ++x)
   data.SetEl(row, column++, 1);
      ++row;
   }
   for (i=0; i<ntemps; i+=4) {

      // Do one row of 4 templates
      for (y=0; y<ydim; ++y) {

 // Do left size border for one row
 column=0;
 for (x=0; x<border; ++x)
      data.SetEl(row, column++, 1);
 // Do one line from the four templates
 thisrow = (ntemps < (i+4)) ? ntemps : i+4;
 for (j=i; j<thisrow; ++j) {
    // do a row
    for (x=0; x<xdim; ++x) {
```

```
       data.SetEl(row, column++, templates[j]->GetEl(y,x));
    }
    // make vertical border
    for (x=0; x<border; ++x)
 data.SetEl(row, column++, 1);
 }
 ++row;
      }
      for (y=0; y<border; ++y) {
 column=0;
 for (x=0; x<xsize; ++x)
      data.SetEl(row, column++, 1);
 ++row;
      }
   }
   data.MakePBM(filename, 1, 1, comment);
//    sprintf(file2, "%s.asc", filename);
//    data.MakePBM(file2, 1, 0);
}



void group::MakeMaster(void)
{
   int i, x, y;

   if (master==NULL)
master = new btemplate(ydim, xdim);
   master->Clear();
   for (i=0; i<ntemps; ++i) {
      for (y=0; y<ydim; ++y) {
 for (x=0; x<xdim; ++x) {
    if (templates[i]->GetEl(y,x))
 master->SetEl(y,x,1);
 }
      }
   }
}


int group::CountBits(void)
{
   int x, y, count=0;

   MakeMaster();
   for (y=0; y<ydim; ++y)
for (x=0; x<xdim; ++x)
     if (master->GetEl(y,x))
  ++count;
   return count;
}

void group::PrintParams(void)
{
   int i;

   printf("\nParamters for the group:\n------------------------\n");
   printf("Template Size:  %d x %d\n", ydim, xdim);
   printf("Bits/Template:  %d\n", onbits);
   printf("Order:          %d\n", order);
   printf("Iterations:     %d\n", iterations);
   printf("Coefficients:   a = %4.2f\n", params[0][0]);
   for (i=1; i<order+1; ++i)
printf("                b(%d,1) = %5.2f \tb(%d,2) = %5.2f\n",
```

```
       i, params[i][0], i, params[i][1]);
}
```

C.2 Header Files

C.2.1 globals.h

```
#define MAXORDER 2

typedef int element_t;
```

C.2.2 matrix.h

```
#include <stdio.h>
#include "globals.h"


////////////////////////////////////////////////////////////////////////////
// matrix class
////////////////////////////////////////////////////////////////////////////
class matrix {
   element_t *data;
   int xdim, ydim;

 public:
   matrix(int, int);
   matrix(char[]);
   void Clear(void);
   int GetXsize(void) {return xdim;};
   int GetYsize(void) {return ydim;};
   void SetEl(int y, int x, element_t) ;
   element_t GetEl(int y, int x);
   int InRange(int y, int x) {return (y>=0) && (y<ydim) && (x>=0) && (x<xdim);}
   void Scale(double);
   void Scale(int);
   void Show(void);
   void Show(FILE*);
   void MakePBM(char*, int, int, char[]);
   ~matrix(void);
};


////////////////////////////////////////////////////////////////////////////
// template class
////////////////////////////////////////////////////////////////////////////
class btemplate : public matrix {
   double b[5][2];
   int order;
   int dim;
   int nchanges;
   int onbits;

   void Initialize(int, int, double[5][2]);
   int Pick2(int []);
   void Switch(int []);
   double Ratio(int []);
   double CalcT(int []);
   void Center(void);
```

```
 public:
   btemplate(int, int);
   btemplate(char[]);
   ~btemplate(void);
   void Generate(int, int, double [5][2], int);
   void PrintLine(FILE*);
};


////////////////////////////////////////////////////////////////////////////
// group class
////////////////////////////////////////////////////////////////////////////
class group {
   btemplate **templates;
   int ntemps;
   int xdim, ydim;
   int onbits, order, iterations;
   double params[5][2];
   btemplate *master;

 public:
   group(int, int, int);
   void MakeTemps(int, int, double, double, double);
   void MakePBM(char[], char[]);
   int CountBits(void);
   void MakeMaster(void);
   void PrintParams(void);
   ~group(void);
};
```

C.3 Sample Parameter File

C.3.1 params.txt

```
16 16 Y x X
10 iterations
-8.0 4.0 8.0 a
0.0 0.4 4.4 b(1,*)
0.0 1.0 0.0 b(2,*)
```

Bibliography

[1] Michael A. Rencher. A comparison of FPGA platforms through SAR/ATR algorithm implementation. Master's thesis, Brigham Young University, December 1996.

[2] M. J. Wirthlin and B. L. Hutchings. Improving functional density through run-time constant propagation. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 86--92, Monterey, CA, February 1997.

[3] J. Villasenor, B. Schoner, K. Chia, and C. Zapata. Configurable computing solutions for automatic target recognition. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 70--79, Napa, CA, April 1996.

[4] Mike James. *Pattern Recognition*. BSP Professional Books, 1987.

[5] Altera Corporation. *Data Book*, 1996.

[6] David A. Patterson and John L. Hennessy. *Computer Organization and Design*, chapter 4, pages 219--221. Morgan Kaufmann Publishers, San Mateo, California, 1994.

[7] R. J. Petersen and B. L. Hutchings. An assessment of the suitability of FPGA-based systems for use in digital signal processing. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications*, pages 293--302, Oxford, England, August 1995. Springer.

[8] Anil K. Jain. *Fundamentals of Digital Image Processing*, chapter 2, 6, pages 33, 210. Prentice Hall, Englewood Cliffs, NJ, 1989.

[9] George R. Cross and Anil K. Jain. Markov random field texture models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(1):149--162, January 1983.

[10] J. M. Arnold, D. A. Buell, and E. G. Davis. Splash 2. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316--324, June 1992.

[11] Michael Rencher and Brad L. Hutchings. Automated target recognition on SPLASH-II. In *Field-Programmable Custom Computing Machines*, April 1997.